



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 file - Votre référence

Votre file - Votre 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

MULTIPLE VIEWS FOR INTEGRATED CASE ENVIRONMENTS

by



Yuchen Zhu

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

Department of Computing Science

Edmonton, Alberta
Fall 1993



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

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 0-315-88143-7

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Yuchen Zhu

TITLE OF THESIS: MULTIPLE VIEWS FOR INTEGRATED CASE ENVIRONMENTS

DEGREE: Masters of Science

YEAR THIS DEGREE GRANTED: 1993

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 rights 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) . . .  . . .

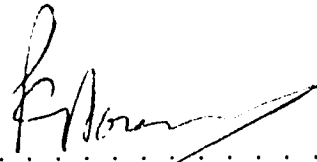
Permanent Address:
Zhejiang Teacher's College for Minorities
LiSui, Zhejiang
China 323000

Date: Jun, 28, 93

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

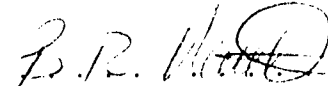
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled MULTIPLE VIEWS FOR INTEGRATED CASE ENVIRONMENTS submitted by **Yuchen Zhu** in partial fulfillment of the requirements for the degree of **Masters of Science**.



.....
Supervisor: Dr. P. G. Sorenson



.....
Examiner: Dr. T. Ozsu



.....
External: Dr. B. Nault (Faculty of Business)

Date: July 21, 93

To my wife, *JianHui Li*

Abstract

The thesis addresses problems of supporting multiple views in a computer-aided software engineering (CASE) metasystem called Metaview. Metaview is aimed to generate automatically software development environments from specifications written in a meta language called EDL/ECL (Environment Definition/Constraint Language). After an environment definer describes his environment model and tool-specific information, a database engine and a tool set are configured to the required environment. Since a system developer may select from many software engineering methods to perform development activities, an environment definer has to model these methods separately in different specification environments. As a result, these environments work rather independently, and, at present, only interact through specialized environment transformation features. However, it is desirable to have integrated environments in which tools share data and communicate in such a way that any changes made in specifications in one environment can be seen immediately in another environment.

A view-based approach to tool integration is proposed in this thesis. It is intended for horizontal tools to take advantage of abstraction, encapsulation and data sharing provided by the multi-view mechanism. A specification environment is said to have multiple views when it is defined as a collection of views that capture both the structural definition and operational semantics of a software engineering method. Issues addressed in this thesis include representation of views in a meta data model and management of specification environments through views. The Entity-Attribute-

Relationship-Aggregate (EARA) data model of the Metaview system is adapted to define “views”. Data sharing is supported through view merging and automatic control transfer is achieved using rule-based operation mappings. Tools within an environment communicate with each other through views, and are thus integrated in terms of their data and control.

As future research areas, the view-based horizontal integration developed in this thesis has to incorporate other types of integration that involves user interface presentation and software processes. The vertical integration of tools that serve different life-cycle phases also need to be supported.

Acknowledgments

I would like to take this opportunity to express my sincere gratitude to those who have helped me throughout my graduate studies. I am indebted to Prof. Paul Sorenson, my supervisor, who taught me everything from how to spell correctly to how to conduct research. He read the thesis draft so many times, and gave me all the guidance whenever needed.

I would like to also thank other members of the Software Engineering Research Group, Dr. Germinal Boliox, Dr. Piotr Findeisen, Narendra Ravi, Lettice Tse and Jesse Lee. Their insightful comments and discussions have been always very helpful.

I am also grateful to my examination committee, Dr. L.Y. Yuan, Dr. M.T. Ozsü, and Dr. Barry Nault for their time and helpful comments.

Finally, I would like to thank my loving wife, Jianhui Li. To support my study, she started working ten days after she joined me in Canada. Without her encouragement, this thesis would not be possible.

Contents

1	Introduction	1
1.1	Basic Definitions	3
1.2	Motivations and Objectives	6
1.3	Thesis Overview	8
2	Background	11
2.1	An Example Two-View Environment for Structured Analysis	12
2.2	Why Supporting Multiple Views Is Difficult?	14
2.3	Basic Concepts of Tool integration	15
2.4	Techniques of Tool Integration: An Overview	19
2.4.1	Data Integration	19
2.4.2	Control Integration	22
2.4.3	Object-Oriented Integration, Combining Data and Control	26
3	Metaview, A System for Generating Environments	29
3.1	An Overview of Metaview	30
3.2	EARA Model	32
3.3	Definitions of Specification Environments	34

3.3.1	Environment Definition Language	34
3.3.2	Environment Constraint Language	37
4	Definition of Views	39
4.1	View Hierarchy	40
4.2	Definition of Primitive Views	43
4.3	View Merging	47
4.4	Composite Views	50
4.5	View Merging Revisited: the Algorithm	55
4.6	Comparison with Other Views	63
4.6.1	Relational Views	64
4.6.2	Object-Oriented Views	65
5	Specification of View Operations	67
5.1	Operation Mappings	67
5.1.1	The Database State	68
5.1.2	Consistency-Preserving Operation Mappings	69
5.1.3	EXA Rules	71
5.2	Defining EXA Rules: an Example	76
5.2.1	Assumptions	76
5.2.2	Events in DFD/CFD Environment	78
5.2.3	Operation Specifications in DFD/CFD Environment	78
6	Integration of Multi-View Environments	82
6.1	Comparison of Integration Paradigms	82
6.1.1	Review of Two Traditional Integration Paradigms	83

6.1.2	A View-Based Integration Paradigm	85
6.2	The View-based Metaview Architecture	88
6.2.1	The View Translator	90
7	Conclusions and Future Work	93
7.1	Thesis Summary	93
7.2	Contributions Of The Thesis	96
7.3	Future Work	97
	Bibliography	100
A	Description of a Two-View (DFD/CFD) Structured Analysis Environ- ment	105
B	Definition of an Integrated DFD/CFD Environment	108
B.1	The Base Views for Structured Analysis Environment	109
B.2	The DFD Primitive View	111
B.3	The CFD Primitive View	113
B.4	The Composite View for the DFD/CFD Environment	115

List of Figures

2.1	The NIST/ECMA Reference Model	16
3.1	Metaview Architecture	31
3.2	Inheritance Hierarchy for the Supertype <i>universal</i>	36
4.1	Model View Hierarchy for Semantic Modeling	41
4.2	Examples of Two Decomposition Approaches	44
4.3	Multiple Views of Data Objects	49
4.4	Coalescing Multiple Views	51
4.5	View Merging Vs. Schema Integration	56
5.1	Consistency-Preserving Operation Mapping	70
5.2	Deletion of an <code>in_out</code> Relationship Object	81
6.1	Data-based Integration Paradigm	83
6.2	Message-based Integration Paradigm	85
6.3	View-Based Data Integration	86
6.4	View-Based Control Integration	87
6.5	Multi-view Architecture of the Metaview System	89
6.6	Software Specification Development	92

List of Tables

3.1	Four Classes of Constraints	37
4.1	Functions/Procedures For Merging Process	58
4.2	TYPE and ATTR for DFD/CFD Environment	63
5.1	Database Manipulation Actions	74
5.2	Events in DFD/CFD Environment	79

Notations

The following are the notational conventions used in this thesis.

- EDL/ECL keywords are denoted in CAPITAL letters.
- Built-in attribute names and data types are denoted in **Sans Serif** font. An exception is when an attribute name appears in a user-defined identifier (e.g. **name** in **data_store.name**). In this case, the **typewriter** font is used.
- Attribute values are denoted in the thesis text font, but enclosed in single quotes, such as 'detects'.
- All user-defined type names, data objects, etc. are denoted in the **typewriter** font.
- The *italics* font is used for emphasis, as well as function or set names that appear in mathematic or logical expressions.
- { } denotes a list of one or more items.
- [] denotes an optional item.
- <> denotes a nonterminal symbol in a syntactic description.
- | indicates separate choices.

Chapter 1

Introduction

Software development environments have evolved from traditional programming environments to more sophisticated Integrated Project Support Environments (IPSE) and Computer-Aided Software Engineering (CASE) environments. The IPSE technologies are concerned with building generic environment infrastructure and platform services to support large-scale software development efforts[BM92]. CASE technologies, on the other hand, concentrate on automating software engineering methodologies to help develop software on time and within budget[CNW89, BFW91]. By *methodology*, we mean a systematic set of *methods* that are designed to support a set of related software development activities. For example, Structured Systems Analysis is a software engineering methodology that includes a set of methods based on data flow diagrams, decision trees, etc.

Integrated software development environments can be classified into three categories based on the nature of the tools:

- environments that are loosely-integrated from tools of different natures and are most often provided by different vendors.
- environments that are composed of tools that are written in conformance with certain predetermined conventions on data formats and control protocols.

- environments that consist of tools from the same vendor, which are written specifically for a common environment.

Besides the previous three categories, there is another category representing environments that are based on a common model associated with a metasytem (also known as MetaCASE or CASE shells). Environments in this category contain universal tools generated by a metasytem from formal environment definitions. A metasytem supports software development life-cycle activities by taking advantages of both the software environment technologies and software engineering methodologies. The main goal in using a metasytem is to generate automatically the major parts of software engineering development environments, which support software engineering methodologies. Environments are defined formally by using an environment definition language (EDL). Relationships between the software objects within an environment or across several environments are further defined by using an environment constraint language (ECL). Metaview[STM92], an ongoing project at the Universities of Alberta and Saskatchewan, is one of major projects following this approach. Other examples are MetaPHOR[DD92, SLTM91] at the University of Jyväskylä, Finland and the Socrates project in the Netherlands[VHW91]. A brief overview of the Metaview architecture and its data model will be presented in Chapter 3. A major advantage of the metasytem approach is that it offers a framework on which environments are generated in a uniform fashion and are potentially well-integrated. Such a framework has not been fully explored by current research efforts in an attempt to implement integrated CASE environments. Environments in a metasytem usually work rather independently, and, at present, only interact through specialized mapping features such as those present in the Environment Transformation Language (ETL)[BST91]. Tools in various environments rarely support integrated functionalities, such as sharing data and communicating in such a way that any changes made in specifications in one environment can be seen immediately in specifications in another environment.

The research work presented in this thesis, as a part of Metaview project, attempts to bridge the gap from independently-generated specification environments to tightly-

integrated CASE environments. The main goal is to allow various environment models to be represented formally in view specifications and thus facilitate the dynamic integration of both environment models and tool-specific mechanisms. Three major issues are addressed in this thesis: the representation of views using the Metaview metamodel, the integration of views through object merging and operation mapping, and the management of specification environments through views. The Entity-Attribute-Relationship-Aggregate (EARA) data model of the Metaview system is adapted to define “views”. Each specification environment is associated with a collection of such views, which enable tools defined for the environment to take advantage of the abstraction, encapsulation and data sharing provided by the view mechanism. Environments communicate with each other through views, and are thus integrated in terms of their data and control capabilities. The potential to support uniform user interface and dynamic process interaction is briefly discussed, but is not a major focus of the thesis.

In the remainder of this chapter, Section 1.1 defines several terms that are used extensively both in the thesis and the literature. These terms are *specification environment*, *tool*, *view*, and *horizontal integration*. Section 1.2 identifies the objectives of the thesis. The motivation for achieving these objectives is also provided. Section 1.3 presents an overview of the thesis. The scope of the thesis is then described, and the expected contributions are summarized.

1.1 Basic Definitions

There are a number of key concepts that are used in this thesis. They include environment, tool, view and integration. Because these terms are also used frequently by researchers in CASE environments, they have diverse meanings, depending on the context in which they appear. For example, the Metaview “environment” used in this thesis, to some extent, has a more specific meaning than people would normally use in a more general context, such as integrated CASE environment, software development environment, and so on. The term “tool” can be used to refer to CASE tools, vendor tools, Metaview tools, etc. Also, the term “view” used in this thesis differs from that

normally used in the database literature. It is necessary to clarify these terms so that their more specialized meanings are understood in our further discussion. Distinguishing these terms will also help to describe the scope of the research work presented in this thesis as compared to related work in the literature.

Many people in CASE communities use so-called integrated CASE environment to mean an integrated set of tools that support one or more processes in the software development life cycle. In this thesis, an *environment definition* is a complete and consistent set of specification object types that are defined using the Metaview meta model. Environments generated from such definitions within the Metaview system are meant to provide support for a particular method within a software engineering methodology. Examples are a data flow diagram environment and structure chart environment. Since the Metaview system is mainly focused on modeling software engineering methodologies that support methods for requirement analysis, high-level design and other upstream activities, environments in Metaview system are often referred to as *specification environments*. To be more specific, we have:

Definition 1.1: A Metaview *specification environment* consists of five elements: environment definition, database engine routines, generic tool descriptions, tool functions, and a common user interface.

Of these five elements, database engine routines and generic tool descriptions are shared by various environments. The user interface is usually designed in accordance with certain window standards, and therefore, also remains essentially the same for different specification environments. Unique to each environment are its definitions written in EDL/ECL, and tool functions that manipulate object types given in the definition. Once a definition for an environment is given, objects that are inherent to the corresponding software engineering method are fixed. To achieve the flexibility of multiple environment definitions for the same specification environment we will incur many integration-related issues as will be discussed briefly in Section 1.3.

A tool is a major component of a Metaview specification environment and is defined as follows:

Definition 1.2: A *tool* in Metaview is composed of two parts: generic descriptions that are to be instantiated by the definition of a specification environment, and tool functions that perform related operations.

In Metaview, there has been little emphasis on tools that are developed external to Metaview, or the problem of incorporating such tools into a Metaview generated environment. The focus has been primarily on generic (or universal) tools that provide operational mechanisms for a specification environment and complement other parts of the environment, namely, the methodology modeling and database management. An example of such a tool is Metaview's Graphical Editor (MGED)[Fin93a] that provides the graphical editing capability for any specification environment defined with the Metaview metamodel.

In this thesis, a view is intended to capture both the structural definitions and operational semantics of a specification environment. With multiple views, different user perspectives are reflected by different views, and the associated operations that access the data conceptualized by such views. A more formal definition of a view is as follows:

Definition 1.3: A *view* in Metaview includes a partial or entire description of a specification environment, and a set of operations that manipulate such a description.

A view for a Metaview specification environment can be related to Metaview tools in the following manner. A view consists of two types of information: environment definitions and associated operations. A universal tool's generic description can be instantiated for definitions that represent various subschemas of the environment definitions. Operations defined with respect to a view are mapped directly to functions provided by Metaview tools. A view definition establishes an association between a subschema and a set of operations that are applicable in the view.

Finally, the term *horizontal integration* is introduced:

Definition 1.4: The integration of tools serving similar purposes in the same phase of software development is called *horizontal integration*.

The following example illustrates the need for horizontal integration. Suppose a requirement analyst wants to elicit and analyze an client's requirements specification for an information system. There are a range of specification vehicles available for this purpose, including function decomposition diagrams, data-flow diagrams, entity-relationship diagrams and action diagrams. Each of these diagrammatic methods represents a different view of the software requirements. In order to support integrated efforts, these different tools have to be well integrated horizontally.

In contrast to horizontal integration, vertical integration is aimed at integrating tools that support different phases of the software development life-cycle. This issue is largely ignored in this thesis because vertical integration normally involves transformations of specifications developed using different environment definitions. The transformation between environments is the topic in [BST91]. Integrating environments for methodologies that share less similarity, such as integrating specification environments for a structured methodology with an object-oriented analysis or design methodology, is still an unexplored area of research.

1.2 Motivations and Objectives

The motivation for the research work in this thesis has two aspects. First, the provision for multiple views for specification environments is currently lacking and must be defined for the Metaview system. Such multi-view mechanisms allow users to examine a software system from different perspectives and facilitate integrated software development. Secondly, environments generated in Metaview system should work cooperatively with regard to both data and function sharing. An example of such cooperation is that changes to specification data in one view should be made available to other views for the same environment, and thereby ensuring that the relevant information is kept consistent across different views.

These two aspects are interrelated. Software systems may be viewed from different perspectives. Typically, each software developer interacts with the CASE environment

via one or more tools that supports his/her role in system development. A developer may want to examine different parts of the system, or the same part of the system using different tools simultaneously. Each tool presents a user with part of the software system being developed. Therefore, integrating these tools will support multiple views of a Metaview environment that is used to develop a software system. Chapter 2 will show that the problem of achieving multiple views of a metasystem environment is essentially the same as integrating CASE tools that are supported in the Metaview system.

In our search for a solution to integrating multiple views of a metasystem environments through tool integration, two important issues need to be addressed:

- *Multiple Representations:* An environment definer should be allowed to define a software object type from multiple viewpoints that correspond to different user roles. For example, a `data_store` type defined for the data flow diagram may have two different sets of attributes associated with two different groups of people, e.g. administrative people and requirements analysts.
- *Dynamic Behaviors Modeling:* Given that a specification database only provides static aspects of software objects used to describe specification information, an environment definer should be able to define the dynamic aspects of software objects. For example, an environment definer may define how and when to invoke operations in one view automatically in response to changes made to data objects in another view of a specification environment. A typical need for such operation mappings arises when consistency needs to be maintained among different representations of the same data object.

In order to address these issues specifically, it is our objective to examine how a metamodel can be used to model and define multi-view specification environments. Such an environment includes not only the basic methodology modeling and tool-specific information, but also the description of possible views that reflect both the multiple representations of software object types and the dynamic behaviors associated with such object types.

1.3 Thesis Overview

Since the primary goal of the thesis is to provide dynamic multi-view support for an integrated CASE environment using a metasystem approach, only those environments that are generated more or less uniformly from declarative specifications in the Metaview system are to be considered for multiple view support. The integration involved is of a horizontal nature, rather than vertical.

Although there are at least three aspects associated with integrating Metaview environments, i.e. methodology modeling, database support and user interface presentation, this thesis will investigate mainly the problems of integrating different aspects of a specification environment represented as model views. Therefore, we primarily examine how different views that are defined to form an environment impact on data sharing and operation mappings. The specification database aspects of the integration are addressed only in terms of the support needed for merging objects represented in different views. How to manage the database effectively to allow for efficient information retrieval and to perform operations on selective objects are beyond the scope of the thesis.

Finally, user interface integration and software process integration are not addressed in any detail and are also considered beyond the scope of the thesis.

The thesis is organized as follows. Chapter 2 elaborates on the basic integration requirements for an integrated CASE environment. A more precise definition of tool integration is given. A data flow diagram environment with two views is presented as an example to illustrate the difficulties of supporting multiple views with the current metasystem approach. By comparing the thesis work with the NIST/ECMA reference model for tool integration[CN92], the problem of supporting multiple views in a metasystem is related to that of the tool integration. Two major tool integration approaches taken by current research efforts are then reviewed: data sharing and message passing.

Chapter 3 includes a brief overview of the Metaview project. The emphasis is on the meta data model and the ECL/EDL languages. The syntax of these two languages are illustrated through examples.

Chapter 4 describes the notion of a model view and how it is used to capture the basic concepts of a software methodology. In order to provide multiple view support to a specification environment, model views are further divided into primitive views and composite views. An environment definer first identifies a set of primitive views that contain basic EDL/ECL statements and defines the object types and possible decomposition schemes that are imposed by a software methodology. The definer then merges these primitive views to form composite views and defines view operations required for consistency checking and control transfer between different parts of a specification environment. By packaging composite views and other related views, the modeled specification environment is defined in terms of views.

Various views defined in Metaview are integrated in two aspects. Multiple representations of data objects are achieved through static view merging, as discussed in Chapter 4. In Chapter 5, the operational aspect of the views defined for a specification environment are examined. Operations specified with regard to certain views have to be mapped to basic tool functions or operations defined in other views. The purpose of such mappings is to assist in the automatic maintenance of the consistency of a specification database and to provide a means of defining user-required operations. The mechanism designed for operation mappings is based on Event-conteXt-Action (EXA) rules. The definitions of the database state and consistency-preserving operation mapping are given in order to provide the basic criteria for the operation mappings.

Based on the view mechanism established in Chapter 4 and 5, Chapter 6 examines further two approaches of tool integration in current commercial systems: integrating data through the use of a common repository or encyclopedia, and integrating control through message passing or broadcasting. A view-based integration framework is then proposed. In this framework, data integration and control integration are tightly-coupled, and treated simultaneously.

The Metaview approach presents a promising framework/platform for tool integration for two reasons. First, each environment is formally defined with an environment model and tool-specific information which are stored in a central environment library.

This allows integration to take place without eliciting further environment information from tool writers that may be ambiguous or vague. Secondly, automatic generation of the environment helps to insure a uniform user interface across tools. Taking advantage of both the view mechanism and Metaview's architecture, the view-based integration framework makes data sharing possible by permitting tools for an environment to operate on a shared database through a collection of views. Control integration between different views of an environment is achieved using rule-based operation mappings that are also defined on the views. Based on this framework, several enhancements to the Metaview architecture are suggested to accommodate multiple views. Central to the revised architecture is a view translator that acts as a general-purpose controller coordinating various environment views.

Finally, Chapter 7 summarizes the research contributions, and discusses possible future research topics.

Chapter 2

Background

The requirement for significant improvements in productivity in the software system development process has motivated substantial research interest in more sophisticated software specification environments. To achieve this goal, tools that are developed for a given software engineering methodology should be well integrated to support diverse specification and design perspectives.

In this chapter, a two-view environment for structured analysis will be introduced. This environment will serve as an example to illustrate the difficulties in supporting multiple views with the current metasystem approach. These difficulties are then related to problems that tool integration can solve. This explains why supporting multiple views in the Metaview system is similar to achieving, in a broader sense, tool integration of CASE specification environments. After reviewing some basic requirements for an integrated CASE environment, a more precise definition of tool integration is given. The definition is based on the integration reference model being developed by the National Institute of Standards and Technology and European Computer Manufacturers Association[Nat91], and the work of others on CASE tool integration[Was90, TN92, CN92]. How the thesis research relates to the NIST/ECMA framework is also described. The last part of this chapter provides an overview of two important aspects of tool integration: data integration and control integration. These aspects also form the basics for two major

approaches taken by current research efforts in integrating CASE environments. PC'TE's Object Management System (OMS) is used as an example to show how tools are integrated by using an object-oriented database. HP SoftBench's Broadcast Message Server (BMS) is discussed to exhibit how tools can cooperate with each other through message passing. The chapter is concluded with a description of research work that combines these two aspects in an object-oriented approach.

2.1 An Example Two-View Environment for Structured Analysis

In this section, a specification environment for structured analysis with two views is introduced briefly to illustrate the difficulties involved in support for multiple views in the current Metaview system. This example environment is referred to as *DFD/CFD Environment* throughout the thesis. Its complete description is included in Appendix A. The purpose is to further motivate and identify the thesis problems, and explain how issues on supporting multiple views are related to integration of CASE tools. This example will also be used in later chapters for discussions on view definitions and integration.

In our example, a data flow diagram (DFD) provides one of the two views for structured analysis. It is used by system analysts to represent system specifications by depicting the flows of data in the system. [GS79] and [DeM79] describe two common styles of representing data flow diagrams. The DFD method used in the following example is adapted from these two styles and is referred to as V_{dfd} . Using Metaview's terminology¹, this DFD view can be described by the following specification schema:

V_{dfd} :

```

AgT = {top_level_dfd, process_exploded}
ET  = {process, data_store, data_flow, terminator}
RT  = {sends, stores, changes, reads,
       derived_from, has_expansion, has_parent_boundary,
       has_child_boundary, has_subparts}
RN  = {source, data, destination, store_name, process_name,

```

¹The meanings of the set names used in the specification schema are explained in Chapter 3.

```

        derived_agg, source_agg, parent, child,
        boundary, superpart, subpart}
AN = {identification, function_description, physical_loc,
      ref_id, access_type, number_of_copy,
      primary_key, form, flow_type, frequency}

```

The second view is an extended DFD for developing real-time applications. There are two commonly-used “extensions”, developed respectively by Ward and Mellor[WM85] and Hatley and Pirbhai[HP87]. In the following example, the Hatley and Pirbhai method is adopted and incorporated with the State Transition Diagram (STD). In Hatley and Pirbhai method, control information is represented separately in a Control Flow Diagram (CFD). The CFD also contains basic concepts such as process, data_store and terminator, but control flows are shown rather than data flows. Another concept that is not present in the DFD is the control specification, represented graphically as a vertical bar in Hatley and Pirbhai’s original notation. A control specification describes how processes are activated as a consequence of events. The STD is used to represent a control specification. This view is referred to as V_{cfd} . A specification schema for this CFD view is given as follows:

V_{cfd} :

```

AgT = {top_level, process_explosion, std_agg}
ET = {process, data_store, terminator, ctrl_flow, ctrl_spec,
      state, event, action}
RT = {in_out, access, has_std, has_sub_std, transition,
      derived_from, has_expansion, has_parent_boundary,
      has_child_boundary, has_subparts}
RN = {input, process_name, output, cspec_name, cflow_name,
      from_state_name, to_state_name, event_name, action_name,
      std_name, sub_std_name, derived_agg, source_agg,
      parent, child, boundary, superpart, subpart}
AN = {form, index, flow_type, state_type, event_type, access_type}

```

The two views given in this section seem to differ only in the data representations, however, this only accounts for part of the differences. The operational part of the two

views may also differ. For example, each view may define its own set of operations that perform view-specific functions in terms of object types defined in the view. The constraints applied to the entities and relationships in the two views may differ as well. This may result in conflicts when checking specification consistency with regard to different views. More details will be given in Section 4.4 where complete definitions of these two views are presented to illustrate how to form a composite view through view merging.

2.2 Why Supporting Multiple Views Is Difficult?

Given the two views described in the last section, the current Metaview approach has two difficulties in supporting both views in the same specification environment for structured analysis.

The first difficulty is related to the static aspect of the environment definition. It is obvious that two sets of representations cannot be present in the same environment definition. For example, if a **process** object is defined as having three attributes: **identification**, **function description**, and **physical location**, it cannot be defined once again as having only intrinsic attributes. Also, if the **form** attribute in a **data store** is defined of string type with limited length, it cannot be redefined to be of **text** type with virtually unlimited length and have both definitions recognized. This also makes information sharing impossible. For example, although the **form** attribute of a data store object is defined differently in the two views, they probably would contain the same information in a specification for a given object. If two views are supported by two separate environment definitions, a redundant copy has to be maintained by the system. One of the advantages provided by the multiple representations is that software objects can be viewed from different perspectives and levels of details. This capability is often provided by environments that support information hiding. Information hiding should be of finer granularity than on a basis of either all or none. For example, a **data_store** defined in V_{cfd} provides an outline view of a data store object, whereas in V_{dfd} , a more complete description of the data store is available.

The second difficulty is that even if two separate environments are generated to represent different user views, there is currently no dynamic connections between the two. Any changes made to the **description** attribute of a process in V_{dd} are not automatically propagated to the **description** attribute of the process defined in V_{cd} . Operations that access the **form** attribute are not shared by the two environments, because of the different data types used to define the attribute. In order to support dynamic operation invocation and automatic operation mappings, the description of a control transfer mechanism has to be built into the environment definitions.

Tool integration mainly involves two types of problems: data sharing and control transfer among different tools. The difficulty associated with data sharing is that different tools may have different representations for data, while control transfer is difficult because an integrated environment has to decide what and when operations are needed in one tool in response to operations in other tools.

The same difficulties have to be tackled in providing multiple views for the Metaview system. First, a means of defining different representations that reflects different user's viewpoints must be provided. Object types defined in this way have to be merged within the same specification environment so that data sharing is possible. The consistency of the data objects has to be maintained, and operations specified with respect to one user's viewpoint must be mapped automatically to operations that correspond to other user's viewpoints.

Since a user's viewpoint on software specifications are usually reflected through the tools being used, different tools, when integrated, are said to represent multiple views. This thesis investigates ways to successfully support these multiple views.

2.3 Basic Concepts of Tool integration

In providing full support for software development, an integrated CASE environment must accommodate a range of applications and user roles. Being general-purposed should not, however, limit the possibility of ultimately being application-specific or

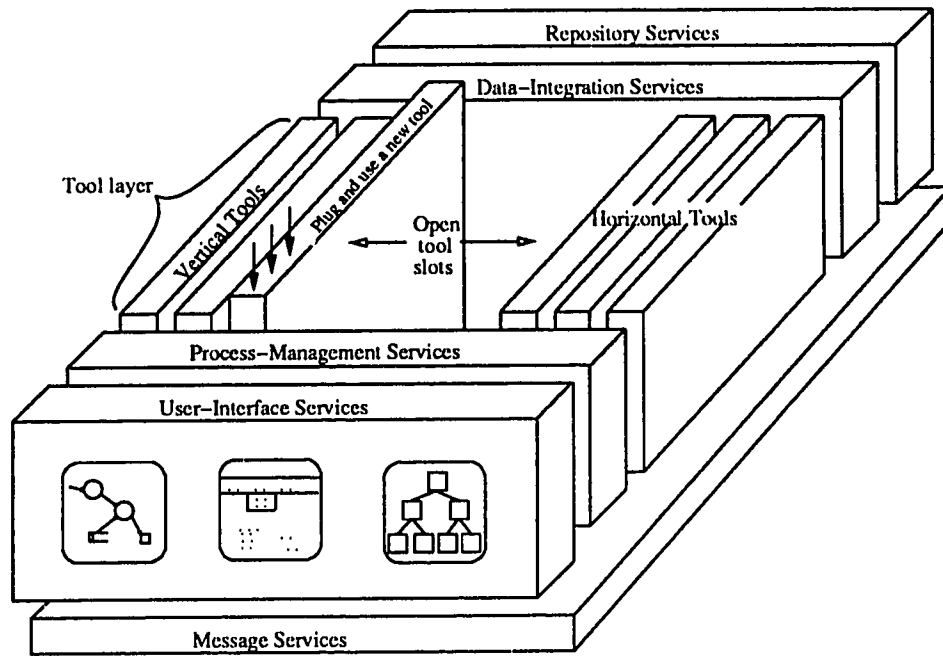


Figure 2.1: The NIST/ECMA Reference Model

user-oriented. Such adaptability should also ease the evolution process in which new methods or tools can be dynamically incorporated. In order to describe the technical aspects of integrated CASE environments, many notions of the NIST/ECMA reference model[Nat91] are used in the following discussion. The NIST/ECMA reference model, as shown in Figure 2.1, has been increasingly adopted as a basis for discussing integration issues by the CASE community.

Tools in the NIST/ECMA reference model are divided into two categories. *Vertical* tools ensures the completeness and consistency of information generated across various life-cycle phases. *Horizontal* tools maintain the integrity of specification information generated within each life-cycle phase when many modeling methods are used.

Services defined in the reference model enable four types of integration: data integration, control integration, presentation integration, and process integration. These

four types of integrations, first proposed by Tony Wasserman[Was90] and elaborated in [TN92] and [CN92] are generally accepted as a basic integration taxonomy. They are briefly defined as follows:

- *Data Integration*: refers to the ability to share software development information (specifications or design information, etc.). It is usually accomplished through direct data transfer, intermediate file and/or shared data repository.
- *Control Integration*: refers to the ability to share functions and to activate other facilities automatically upon data changes, occurrence of events or user intervention. The control integration mechanisms include explicit message passing, integration management (i.e. control daemon), rule-based triggers and a broadcasting message server.
- *Presentation Integration*: refers to the uniform appearance and behavior of the environment user interface. This is often done by building an integrated CASE environment on the basis of some commonly-used window standards, such as OSF/Motif and SUN/Open Look.
- *Process Integration*: refers to the ability to invoke the right facilities at the right time by the right personnel in support for the software process that fits into an organizational or project model. This integration is usually accomplished through modeling life-cycle phases, planning project management and using metrics-based quality control.

In the NIST/ECMA reference model, data integration is supported by the data-integration and repository services. Control integration mechanisms include explicit message passing, time- or event-activated triggers, and message servers. To achieve control integration, tools must be able to notify each other of events, activate other tools and share functions.

All these four types of integrations should be addressed when designing an ideal integrated CASE environment because they are all interrelated. For example, control

transfer is often caused by changes to the data, which may in turn affect the integrity of the data across tools. Also, the tool invocation mechanism, which is a control integration aspect, provides fundamental support for process integration. This interrelationship is often overlooked by many researchers, and therefore, the integration strategies proposed so far are often directed to only one of the issues. For example, the broadcasting message service technology is only concerned with the problem of control integration while ignoring data sharing issues, whereas an object management system is aimed at the support for data integration and may ignore aspects of function sharing. As a result, tools developed ignoring these important inter-relationships are doomed with respect to good performance because they are unable to utilize fully the resources available or underlying technologies related to data repositories, and software engineering methodologies. Some integration efforts based on an object-oriented paradigm are trying to address both the data and control aspects of the integration at the same time, as will be briefly discussed in Section 2.4.3. In this thesis, we will bring these two aspects together in a metasystem approach that differs from a purely object-oriented one. The focus is on the data and control integration.

It is assumed that the Metaview system is responsible for providing the metamodel, query and data-interchange services. The research in this thesis will lead to a framework for tool integration in the context of the Metaview system. This framework is based on a multi-view mechanism and an event-driven operation mapping mechanism. Contrary to conventional database views and program views, a view of a Metaview specification environment not only refers to the virtual description of a common specification database, but also to the dynamic behaviors associated with a set of tool functions that manipulate such descriptions. The purpose of operation mappings is to assist in automatic maintenance of the consistency of a specification database and to provide a means of defining user-required operations with regard to his/her specific views.

These two forms of integration are important in providing multiple view support for horizontal tool integration in the Metaview system. In Metaview, support for vertical tool integration is achieved using transformations specified in ETL (Environment Transformation Language)[BST91, Lee92].

2.4 Techniques of Tool Integration: An Overview

Tool integration has been an active research area during the past few years[Dow87, Tho89b, Was90]. There are two reasons for this trend. First, the diverse nature of the modern applications have been the major driving forces for many new development methods. More powerful tools are required to support adequately these new methods. It has been clear, however, that no single tool can meet the expanding needs for automated support in information-based applications development. Secondly, recent efforts by the CASE community have produced some CASE standards and frameworks that further encourage the integration efforts. The examples are ECMA PCTE[Eur90], ANSI's Information Resource Dictionary Systems[ANS88], Electronic Industry Association's CASE Data Interchange Format[Ele90] and IRDS ATIS proposed by Atherton Technology[CAS91]

In the next three subsections, previous work on data integration and control integration is reviewed. We believe these two forms of integrations are the two most important aspects of tool integration and are our major concerns in this thesis. Issues related to presentation and process integrations will be left largely as open problems for future research. Since our objective is to bring these two forms of integration together, some combined efforts using object-oriented approaches are briefly reviewed.

2.4.1 Data Integration

Data integration is one of the two major approaches to achieving cooperation among tools. In an integrated CASE environment, the data represent a description of the application system under development, including project plans, requirement specifications, design documents, program modules, test data and so on. The goal of the data integration is to maintain consistent information and make them accessible to various tools. Thomas and Nejme[TN92] define five properties related to how well tools agree on the way data are manipulated. These properties are interoperability, nonredundancy, data consistency, data exchange and synchronization, with the first three related to persistent

data, and the latter two to nonpersistent data. Based on the granularity of the data to be shared, data integration can be divided into two categories[HKO92]: coarse-grained integration, as exemplified by shared files, and fine-grained integration, as exemplified by textual lines or symbols describing software objects.

The traditional strategies for data integration are coarse-grained. When the data formats used by two tools happen to be the same, they are transferred directly. The UNIX pipe is an example of a mechanism that supports this kind of transfer. When the formats differ, certain conversions are carried out before the files can be shared by the two tools. Some intermediate data format is often present in this case. Examples are CASE Data Interchange Format[Ele90] and AD/Cycle's External Source Format (ESF)[MB90]. File systems are not sufficient to enable multiple tools to work concurrently. Services such as transaction management, concurrency control, version control, data integrity and security should be also provided.

As the recent repository technologies became more sophisticated, many people used a shared repository as the basis for the data integration. Notable examples are PCTE's Object Management System (OMS)[BMT88, Tho89a, TTBG90] and AD/Cycle's Repository Manager[Sag90]. The rest of this subsection discusses PCTE's OMS as an example to see how data integration is supported through a shared repository. The reason for choosing PCTE's OMS is that it is typical of data integration approaches and the one that appears to be receiving the most interest amongst tool vendors.

PCTE is a specification for a set of building blocks from which tools can be built to form environments. It specifies platform services and interfaces which enhance tool integrations. In relation to data integration, it provides OMS facilities for defining and managing data stored and manipulated in an integrated CASE environment.

The OMS data model is based on a modified binary Entity-Relationship model. Entities (or objects in the OMS) are typed with a name, a parent type, a set of attribute types and two sets of link types, of which one serves as the link origin and another the destination. A relationship between objects is established as a pair of mutually inverse links.

Object types form a hierarchy with a single predefined root type called `Object` and a few predefined `Object` child types.

Object, attribute and link/relationship definitions are grouped into Schema Definition Sets (SDSs) which themselves belong to a predefined object type named `sds`. A type definition may be distributed across multiple SDSs while on the other hand, the conceptual model of the PCTE object base is captured by all the SDSs which may be overlapping.

Given these basic features of the OMS, data integration takes place in the following manner. Each tool (or PCTE process) has a working schema which is a dynamic union of SDSs and defines a particular external view of the object base subset that is interesting to the tool. It is “dynamic” because the composition of SDSs can be reconfigured during tool execution. Tools access objects are based on a navigational model defined in the tool’s working schema. The concurrency control and integrity checking are supported through the distributed management of schema information and an OMS *activity* mechanism. Each tool runs within the context of one of three levels of activity: unprotected, protected and transactions.

More recent efforts in evolving PCTE have resulted in a more comprehensive interface specification called PCTE+. The history of this version can be found in [BMT88]. The extensions made in PCTE+ include:

- composite entities, a collection of objects are treated as a single object when being manipulated.
- a self-referential model – the metabase, a part of the object base in which object, link and attribute types defined for the OMS are represented as objects, allowing uniform querying of both schema data and specification data.
- version support, the version management is supported using the idea of composite entities.
- multiple inheritance for object types with naming conflicts resolved by SDS mech-

anism's scope rules.

- security, both discretionary and mandatory access controls are possible.
- a change notification mechanism working in a distributed environment.

These new features contribute to improvements in data integration. The composite entities help model complex software development information more completely and effectively, and version control and access control provide better information management. One of the most difficult problems, schema evolution remains unsolved. Without its solution, the flexibility in adding new tools into the integrated CASE environment is not well supported.

2.4.2 Control Integration

An important requirement for effective control integration is the support for implicit tool invocation, rather than forcing a user to invoke each tool explicitly. Early implementations of control integration focused on coordinating teams through facilities like electronic mail and configuration management. In the context of an integrated CASE environment, control integration may take place between horizontal tools or vertical tools. For example, the entity-relationship diagram tool may have to be invoked when a requirements analyst makes some changes to the data-flow diagram. A transformation controller may invoke a code generator when a system analyst changes requirement information. Control integration should be concerned with providing facilities like function sharing across tools, event notification and tool activation. When combined with software process modeling, control integration can assist to achieve the process integration. In [TN92], authors have identified two properties defined on the control relationship between two tools: provision and use, which represent two opposite directions of function sharing.

There have been numerous mechanisms developed for control integration during the past years. Most notable are those based on explicit message passing, active data,

control daemons, and selective message broadcast. In the rest of this subsection, we concentrate on the broadcast message service (BMS) technology which was first developed by Steve Reiss of Brown University in his FIELD project[Rei90]. This work was later refined and used in HP SoftBench Environment[Cag90]. (SUN ToolTalk and DEC FUSE environments also adopted the similar technology, but will not be covered here.)

Reiss's main idea of integration through message passing is that existing independent tools can be made to communicate directly with each other if they are modified to incorporate so-called *message interfaces*. These interfaces allow tools to register interested message patterns with a message server (*Msg*) and to send and receive messages to and from the *Msg*. Tools interact by sending messages to *Msg*, which re-broadcasts them selectively to those tools whose registered message patterns match the messages being broadcasted.

There are several drawbacks related to Reiss's approach. First, writing message interfaces for tools to be integrated could be a painful experience, because one has to find out what messages are or will be available and what information they convey. Second, once a new tool is added, old tools have to be modified to take advantage of the functions provided by the new tool. On the other hand, once a tool is removed from the environment, other tools also need to be modified to take into account the fact that a tool's functions are no longer available. Third, FIELD does not provide a formal management scheme for either the syntax or the semantics of various messages, let alone a common semantic language to serve as the foundation for effective communication. As a result, tools are very loosely integrated. Finally, multi-user interaction is not supported in FIELD, which is part of the reason that it is only suitable for programming-in-the-small, and not for programming-in-the-large or programming-in-the-many.

HP SoftBench tools communicate in a networked environment via a broadcast message server (BMS), which is very similar to Reiss's *Msg*. As Cagen claims in [Cag90], the HP SoftBench integration architecture is designed to provide mechanisms that support tool collaboration in a distributed computing environment. HP SoftBench has three primary components. The remote tool execution and data management facilities support

collaboration of distributed team work. The OSF/Motif-based user interface management facilities provide the basis for presentation integration. And the most important facilities, the tool communication mechanisms, allow two-way, one-to-many or many-to-one and event-driven control integration. The significant improvements included in HP SoftBench over FIELD are:

- uniform message format
- event trigger and tool execution management
- HP Encapsulator

Messages in HP SoftBench are divided into two types, *requests* and *notifications*. They all have uniform format, specifying various information, such as a tool protocol, specific operations required, the location of the data being operated on, and other optional information needed for the operation.

All tools integrated in HP SoftBench environment announce the actions just taken after each operation performed. This notification message is picked up by other interested tools to trigger the execution of a set of operations that are either predefined in the tools or later defined by users. When a request message is received by the BMS, and there are no running tools servicing this request, the HP SoftBench tool manager starts an appropriate tool to handle the need. This trigger/execution management mechanisms allow automatic and implicit control transfer which is considered to be a favorable feature required by control integration.

HP Encapsulator[Fro90] is a translator of commands, actions and presentation between existing tools and the rest of the HP SoftBench environment. It allows existing tools to be encapsulated in a message-based application program interface (API) so that tools can be linked to HP SoftBench network-wide communication and triggering facilities including the BMS, the event handler, the pattern matcher and the tool manager. It also helps a user modify tool protocols and tailor the HP SoftBench environment to support particular software process needs. The *Encapsulator description language* is used

to simplify the task of describing an encapsulation. This specification language captures the two main components of the tool encapsulation: interfaces and actions. Interfaces are tools' connectors to the HP SoftBench window system and message system. Actions are the steps to be taken when certain conditions are met on a tool's interface.

There are some limitations imposed by the HP SoftBench environment.

- HP Encapsulator's customization flexibility is restricted to events or atomic operations whose behaviors can be intervened from command-line flags.
- Users have complete control over what tools or operations respond to what messages, subject only to the event granularity restrictions. This may not be desirable from an organizational point of view. The tool protocols should be prioritized, giving some people like project managers more power over ordinary team members in controlling the way tools interact with each other.
- Another drawback results from the separation of data integration from control integration. Almost no data sharing is possible in HP SoftBench, which means data manipulated by various tools are normally disjoint. As a result, multiple methods aiming at the same software process, if supported, would be treated as totally independent tools, which obviously is not desired.
- At the present time, HP SoftBench is basically an integrated programming environment. Only a few lower CASE tools have been integrated, such as the program editor, static analyzer, program debugger, program builder and mail. HP Encapsulator promises the customization and extension capabilities of integrating existing tools into HP SoftBench environment without even modifying the source code. It is, however, really doubtful whether HP SoftBench will be effective in integrating upper CASE tools or cross life-cycle tools, because these tools often require sharing data and managing data relationships. Further difficulties may arise when trying to integrate presentation schemes of these upper CASE tools because of their highly interactive, graphical user interfaces.

2.4.3 Object-Oriented Integration, Combining Data and Control

Integrating both data and control simultaneously would presumably take the advantages offered by both. This is indeed a more favorable trend for tool integration because it is an important feature that should be present in an integrated CASE environment.

We briefly look at two examples representing these combining efforts: Object Management Group's Common Object Request Broker Architecture and Specification (CORBAS)[COR91] and IBM's Object-Oriented Tool Integration Services (OOTIS). The question is how to combine data sharing with message servicing techniques to provide both integrations.

The CORBAS[COR91] defines a common architecture and specification to allow the integration of a wide variety of object systems. Central to the CORBAS architecture is an the concept of *Object Request Broker* (ORB). An ORB provides the mechanism by which objects transparently make requests and receive responses. An object in CORBAS is an identifiable, encapsulated entity that provides one or more services that can be requested by a client. An interface of an object describes a set of possible operations that a client may request of the object. Interfaces are specified in *Interface Definition Language* (IDL). An object also has its implementation, which is a set of methods that are to be executed (or activated) in response to services requested.

It is an explicit goal of the Common ORB Architecture to allow interoperation between different object systems and ORBs. ORBs are structured in such a way that it is responsible for all the mechanisms required to find the object implementation for any request sent an object, to prepare the object implementation for the request, and to communicate the data making up the request. The interface the client sees is completely independent of where the object is located, what programming language it is implemented in, or any other aspect which is not reflected in the object's interface.

In order to make multiple object systems interoperable, IDL-defined object-oriented invocation model is employed to allow requests to pass through ORBs, while preserving the invocation semantics transparent to clients and implementations. There are three

possible ways suggested in CORBAS to realize the interoperation of multiple ORBs: *reference embedding*, *protocol translation*, and *alternate ORBs*. With reference embedding, an object in one ORB appears to be an object in the second ORB. An invocation on the second object is actually performed on the implementation in the first ORB. Protocol translation deals with the situation where multiple ORBs differ in their implementation details. Requests in one ORB is translated to requests in other ORBs. Finally, the alternate ORB technique makes the same objects available to multiple ORBs. An object implementation is bound to an object adapter which presents an identical interface to each ORB. Object references are generated in different ORBs. Multiple ORBs are supported in such a way that few changes are necessary to the object implementation code, and object adapters serve as the interfaces to allow different ORBs to obtain equivalent object references.

Some efforts have been made in IBM's OOTIS (Object-Oriented Tool Integration Services)[HKO92] with a fine-grained object-level support. In OOTIS, a software artifact is represented as a large network of interconnected fine-grained objects. Tools for processing such objects are written in object-oriented style — each object belongs to a class that defines not only its representation but also its operations. Messages can be applied to an object through its interface. The code (or methods) to be executed in response to a message is determined at run-time based on both the message itself and the class to which the object belongs. A tool in this context is no longer a single chunk of code. It consists of a collection of methods spread across a number of classes. Consequently, an environment is viewed as a collection of tools, each providing its particular collection of methods and classes. This extended object-oriented approach integrates both data and control through a message dispatch mechanism which automatically maps messages and event calls to appropriate objects. Since all data are represented as objects and organized in an object-oriented database, its performance relies heavily on the object-oriented database management system.

Strictly speaking, integration based on the object-oriented paradigm is still in its early experimental stage. No successful systems have been reported so far. The reason lies with the limitations of the object-oriented data model to model two important

aspects of software engineering methodologies:

- Relationships modeling. This is difficult because both entities and relationships must be modeled as encapsulated objects. In modeling software engineering environments, it is important that an entity needs to “be aware of” the other entities with which it is participating a relationship. This can be modeled rather awkwardly in an object-oriented system by traversing through an intermediate object (relationship). This type of traversal requires a significant operation overhead.
- Cross-tool constraints. Again, due to the strict encapsulation, constraints that define data consistency checking among different tools would impose a problem that neither of the tools can claim the ownership of the constraints.

In part, because of these difficulties, the extended entity-relationship models have become the dominate models used by the software methodology modeling community.

Chapter 3

Metaview, A System for Generating Environments

The discussion in the previous chapter indicated that traditional integration approaches, namely, repository-based data integration and message-broadcasting-based control integration, ultimately encounter difficulties when trying to support tool heterogeneity. Our primary interest in this thesis is to investigate integrated environments that consist of universal tools generated in the Metaview metasystem.

This chapter gives an overview of the Metaview system. The purpose is to establish the ground work for the discussion in the following chapters. Section 3.1 provides an introduction to the Metaview system with emphasis on its three-level architecture. Section 3.2 introduces the EARA metamodel that is used to represent CASE environments. Section 3.3 reviews two major aspects of the Metaview system, the Environment Definition Language (EDL) and Environment Constraint Language (ECL). These two languages are used for expressing the elements of the EARA data model so as to model software engineering methodology embedded in a specification environment. These two languages also serve as the basis of view definitions for a specification environment (i.e., a basis of the proposed language extensions used for defining views for a specification environment).

3.1 An Overview of Metaview

Figure 3.1 provides an architectural overview of the Metaview metasystem. It is simplified from its original version[STM92] in order to emphasize its three-level architecture. The three levels, the meta, environment and user levels, are demarcated by horizontal dash-lines. Shown vertically are three major parts of the Metaview that provide functional support for database facilities, environment definition and tool generation, respectively.

At the *meta level*, the meta definer defines a metamodel for Metaview which is the EARA model, an extension to Chen's E-R model[Che76]. Also at this level, there are two libraries of generic routines that should be categorized and collected. They are tool related routines that are common to all environments, and database engine routines that are used to manipulate the specification database. Generic tool routines include those used for user interaction (textual or graphical editors), report generation and query handling. Database engine routines, currently implemented in Prolog, consist of database initialization and shutdown activities and specification update and query rules.

At the *environment level*, the environment definer describes an environment by using Environment Definition Language (EDL). Constraints that govern the consistency and completeness in specifications and environment definitions are identified and expressed by using the Environment Constraint Language (ECL). Tool-specific descriptions of the environment services that supplement the generic tool information are provided for generating tools of the environment. After an environment definition is processed by the EDL compiler, a set of environment tables (also Prolog facts) are collected in the environment library. These are fed into database engine and tool generators to configure the required environment. The database engine is configured by linking the environment tables to the generic database engine routines through an environment-independent universal interface called Project Daemon.

At the *user level*, a system developer may select a particular development environment to perform requirements analysis, detailed design or other life-cycle activities. For

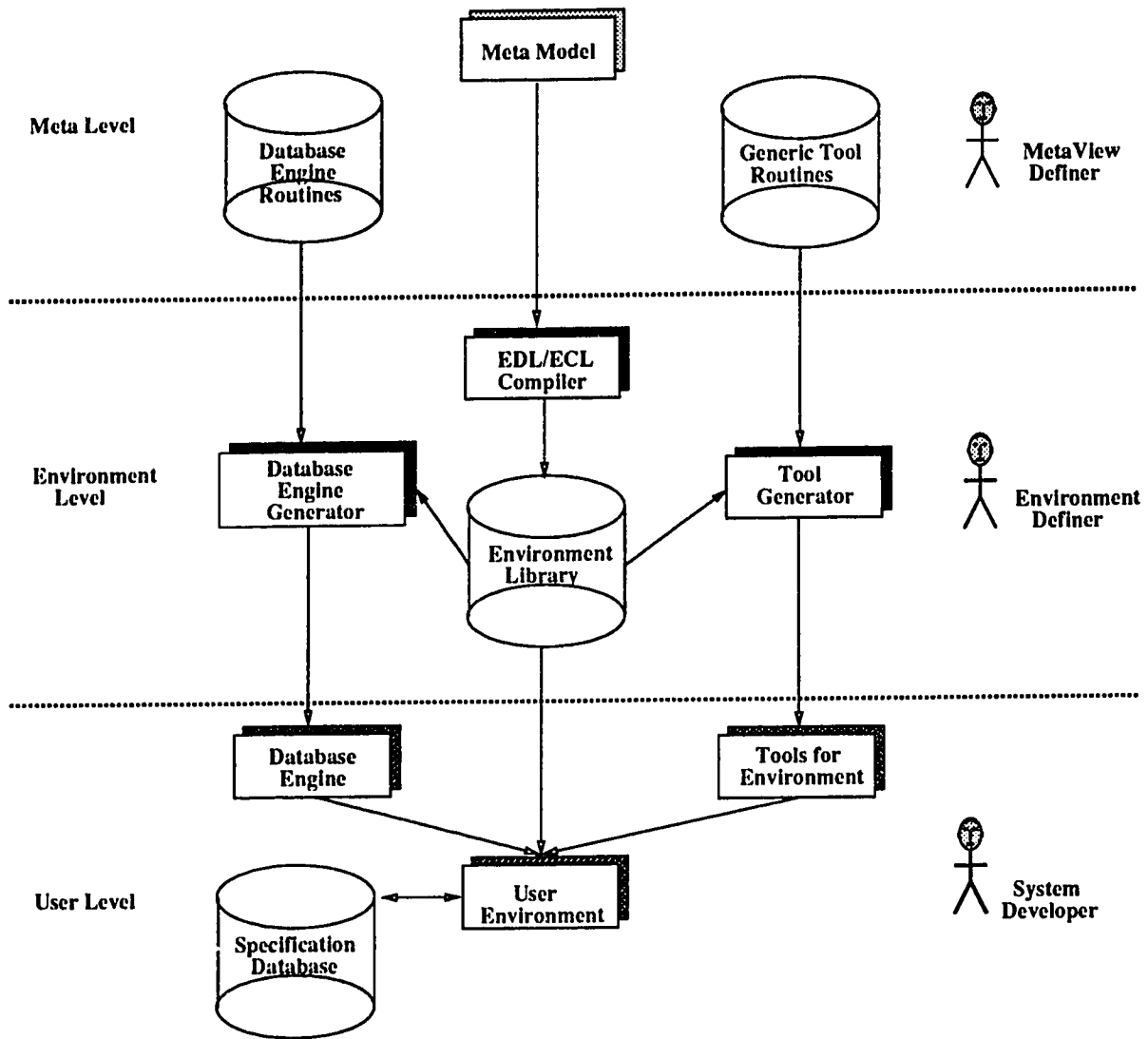


Figure 3.1: Metaview Architecture

example, he may create part of his requirements specification using a data-flow diagram editor provided by a structured analysis environment. He may also examine the specifications produced by querying the specification database or make modifications to the specification.

Although the Metaview's metamodel supports multiple specification environments, its architecture has limited power in this regard. Environment definers are allowed to define any number of specification environments. Currently, environments that are configured according to such definitions would work independently without knowing of each other's existence. Vertical integration is supported using a specification transformation approach[BST91]. Specifications produced in the source environment are transformed to those in the target environment. The transformation process is formalized using transformation rules expressed in the Environment Transformation Language (ETL). However, horizontal integration facilities are not provided to help manage the different views of these environments. As a result, Metaview environments are currently not well integrated horizontally. The transformation between these environments, e.g. from state transition diagram to structure chart[Lec92], is considered to be a vertical integration, which is not discussed in this thesis.

A more complete description of the Metaview architecture can be found in [STM92].

3.2 EARA Model

In modeling a CASE environment, we first need to identify what types of information are to be modeled (i.e. what software objects can participate in a specification). This is described by a *specification schema*. For example, the specification schema for the entity-relationship model defines the notions of entity set, relationship set, and attributes, and how these concepts are related.

To support a software engineering methodology, the EARA model is used to capture the semantics inherent in hierarchies of complex objects. For this purpose, three kinds of abstraction mechanisms are identified in the EARA model. The first is *aggregation* which

combines a finite set of entities and relationships to form a high level aggregate object in support for information hiding and easier object reuse. *Generalization* establishes an *is-a* relationship from a subtype to its supertype in support of object inheritance. The third type of abstraction is *classification* which establishes an *instance-of* relationship from instances to their common type in support of type instantiation.

The EARA (Entity-Attribute-Relationship-Aggregate) model was first proposed by McAllister in his PhD thesis[McA88] and was later refined in [STM92] and [Fin92]. It supports aggregation through the aggregate type that represents a collection of entity and relationship types. It supports generalization through its inheritance hierarchies. The classification capability of the EARA model is simply reflected in EARA database state which will be in Subsection 5.1.1.

To be more specific, the specification schema for the EARA model contains six sets of descriptive elements and eight functions that define the interrelationships between these sets.

<i>AgT</i> :	a finite set of <i>aggregate</i> types	
<i>ET</i> :	a finite set of <i>entity</i> types	
<i>RT</i> :	a finite set of <i>relationship</i> types	
<i>AN</i> :	a finite set of <i>attribute</i> names	
<i>RN</i> :	a finite set of <i>role</i> names	
<i>VT</i> :	a finite set of <i>value</i> types	
<i>ct</i> :	$AgT \rightarrow Powerset_1(ET \cup RT)$	-component type function
<i>ai</i> :	$ET \rightarrow AgT$	-aggregate identification function
<i>aa</i> :	$(AgT \cup ET \cup RT) \rightarrow Powerset(AN \times VT)$	-attribute association function
<i>rm</i> :	$RT \rightarrow Powerset_2(RN \times \{single, list\})$	-role mapping function
<i>rpt</i> :	$RT \rightarrow Powerset_1(Powerset_2(ET \cup AgT))$	-relationship participant types function
<i>ag</i> :	$AgT \rightarrow AgT$	-aggregate generalization function
<i>eg</i> :	$ET \rightarrow ET$	-entity generalization function
<i>rg</i> :	$RT \rightarrow RT$	-relationship generalization function

The Metaview system was originally intended to support early CASE activities, including those of Structured Systems Analysis and Design. Downstream activities such as coding and testing are not well addressed in Metaview, or in metasystems in general [DST86, SM88]. However, even defining specification environments requires extensive familiarity with EDL/ECL and solid knowledge of the methodology to be modeled. Typically, a system analyst would not modify the environments to suit his particular needs unless he asks an environment definer to do this on his behalf.

3.3 Definitions of Specification Environments

There are two major aspects related to defining a specification environment using the EARA metamodel. First, the Environment Definition Language (EDL) is derived from the EARA metamodel to capture the basic concepts inherent to the specification environment and the relationships among these concepts. Secondly, Environment Constraints Language (ECL) is used to define constraints formally to allow for automatic checking of a specification's consistency and completeness.

These two topics are discussed in this section and the dataflow diagram (DFD) environment is used as an example to illustrate the syntax of the two languages. A more complete definition of the DFD environment can be found in [McA88, BST91]. We assume readers are familiar with Structured Systems Analysis methodology. An overview and evaluation of this methodology can also be found in those two papers. Other books such as [GS79, DeM79, YC78] may also serve this purpose.

3.3.1 Environment Definition Language

In defining an environment based on the EARA model, an environment definer usually has to follow four steps:

- Model basic concepts underlying the software engineering methodology and relationships between them.

- Model the hierarchical decomposition strategy imposed by the software engineering methodology through the aggregation feature of the EARA model.
- Specify the constraints that govern the environment integrity by using ECL statements.
- Determine what kind of graphical objects should be presented to the system developers/analysts, and map the EARA definition of the environment to its graphical extension.

EDL in Metaview is a declarative language for defining the basic types of database objects associated with EARA databases. It is used primarily for first two steps. This section gives an introduction to the EDL language through some examples. A full definition of EDL syntax can be found in [McA88]. The ECL is discussed in the next subsection. For the graphical extension, readers may refer to [Sch90] for details.

From the EDL creator's point of view, EDL is just a linguistic embellishment of the EARA model. To illustrate the EDL syntax, a part of DFD environment is modeled as follows:

```
ENTITY_TYPE universal GENERIC
    ATTRIBUTES (description: text);
ENTITY_TYPE data_object GENERIC IS_A universal
    data_flow IS_A data_object
    stored_data IS_A data_object
    process IS_A universal
    interface IS_A universal;
ENTITY_TYPE data_store IS_A universal
    ATTRIBUTES (form: string(1..30));
RELATIONSHIP_TYPE send
    ROLES (source, data, destination)
    PARTICIPANTS
        (process, data_flow, process | interface)
        (interface, data_flow, process)
    ATTRIBUTES (frequency: time_per_unit);
VALUE_TYPE time_per_unit
RECORD
```

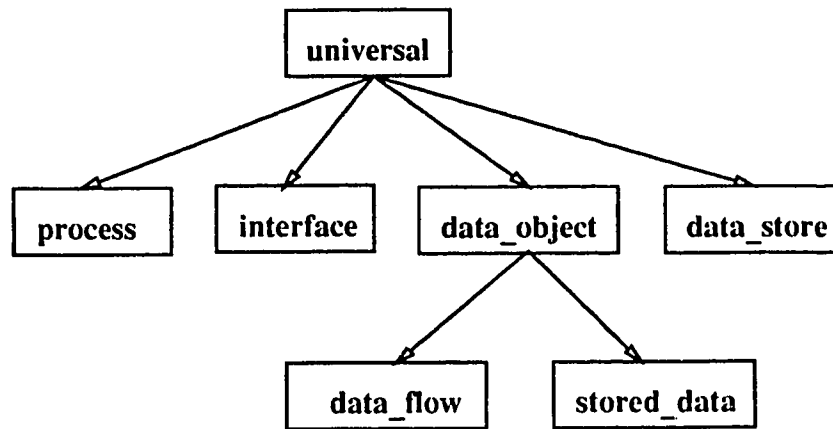


Figure 3.2: Inheritance Hierarchy for the Supertype *universal*

```

    quantity: integer
    unit: (second, minute, hour,
          day, week, month, year);
  END RECORD;
AGGREGATE_TYPE data_flow_aggregate
  COMPONENTS (ALL)
  BECOMES process
    WITH CONNECTION(sends)
  ATTRIBUTES (description: text);

```

The first two lines in the DFD environment definition defines a supertype called *universal*, on which several subtypes are defined. This inheritance hierarchy is shown in Figure 3.2.

ENTITY_TYPE statements define basic concepts of the DFD environment, and RELATIONSHIP_TYPE statement defines a ternary relationship type named **sends** that has three types of entities acting as **source**, **data** and **destination** respectively. The attribute **frequency** is of a user-defined data type called **time_per_unit** which is defined in the next VALUE_TYPE statement. Some other relationship types, such as **access**, **changes**, and **reads**, can also be defined similarly, but are not shown above. The last statement defines an aggregate type that may contain all entity or relationship types

Class Number	Constraint Definer	Information Constrained	Descriptive Phrase
I	Metasystem Definer	Specification Database	Generalized Specification Constraints
II	Environment Definer	Specification Database	Environment Specific Constraints
III	Analyst	Specification Database	Analyst-Supplied Constraints
IV	Metasystem Definer	Environment Definition	Environment Generation Constraints

Table 3.1: Four Classes of Constraints

as its component types, and an dataflow aggregate may be treated as a single process entity if it appears in another aggregate. The hierarchical decomposition of a dataflow diagram is modeled in more detail in [McA88].

3.3.2 Environment Constraint Language

Constraints arise during the creation and use of a Metaview environment. They facilitate the automatic checking of the consistency and completeness of a specification. The ECL does not address the question of how to resolve constraint violation. It simply provides a means for specifying the conditions that a specification must satisfy.

In the context of a metasystem, constraints can be defined by the metasystem definer, the environment definer or a system analyst at all three different levels to constrain different source of information. As a result, four classes of constraints are identified as shown in Table 3.1. This table is copied from [MSTD87] for completeness.

Environment generation constraints and generalized specification constraints are embedded in the EARA model, and enforced in the implementation of the metasystem software. ECL is directed to the specification of environment specific constraints. Analyst-supplied constraints, which are defined as part of a particular specification and

applicable only in the context of that specification are ignored in the construction of Metaview.

We do not intend to get into the details of the ECL notations. A complete description of ECL syntax and some examples can be found in [McA88] and [MSTD87]. The following example illustrates how constraints are defined on `data_flow` objects.

```
CONSTRAINT data_must_be_sent Is
  --- Every data_flow entity must participate
  --- in a sends relationship.
  OBJECTS
    df := (data_flow);
    MUST_HAVE
      (sends: df = *.data);
END;
```

In summary, this chapter established a basis for multiple view support in specification environments generated using Metaview. The Metaview architecture provides a framework for building a horizontally integrated specification environment. The current EDL/ECL will be extended to express views in the next chapter.

Chapter 4

Definition of Views

In this chapter, the formal representation of multiple views within a specification environment will be examined based on Metaview's EDL/ECL. For this purpose, it is necessary to identify what views should be modeled and what information needs to be captured in a view definition. The first section presents a two-level view hierarchy that includes primitive views and composite views. Primitive views are designed to capture conceptually the major parts of an environment. Primitive views are defined and discussed in Section 4.2 using an example from the DFD/CFD environment. How view merging takes place on primitive views is then described in Section 4.3. The merging process is illustrated using the DFD/CFD views of a structured analysis environment. The view merging mechanism establishes a basis for the view-oriented tool integration that is discussed in more detail in Chapter 6.

Primitive views are capable of providing encapsulation for most software engineering methods. A composite view is the level of abstraction formed from merging different primitive views to support an integrated specification environment. The formal representation of composite views is described in Section 4.4.

After the definition of a composite view for the two-view structured analysis environment is given, view merging will be re-examined in Section 4.5, where a detailed merging algorithm is presented.

Finally in Section 4.6, we compare our view-oriented approach with other approaches, namely, the traditional approach of relational views and more recent approaches of object-oriented modeling.

Operation mappings are left to the next chapter where further discussion illustrates how such mappings can help maintain the database consistency and completeness, as well as automatic function invocation across composite views.

4.1 View Hierarchy

Support for multiple views is not a new idea. Some work in this area has been done in developing programming environments. Examples are PECAN by Reiss[Rei85], and a structure-oriented environment developed at Carnegie-Mellon University[Gar87, GM84, CGG⁺85]. In these systems, multiple views are built for editing, compiling and executing programs. For instance, a program may be presented to a programmer in its created form or in a beautified form (e.g., source code is displayed after being processed by UNIX `cb` command). When a program becomes large, an outline view is useful to show all the functions that are included in the program, and help in understanding the program more easily. As the program is compiled, systems also provide several other views, such as a call graph that shows the relationships between the participating functions, and a symbol table that gives the detailed characteristics of all declared variables.

For requirement analysis or design environments, researchers and developers have concentrated on separate CASE tools that are written as software engineering methodology companions[VJT92]. The notion of multiple views does not as yet seem to be clearly defined for CASE tools. At the beginning of the thesis, we defined a methodology to be a systematic approach for software development. In other words, a methodology is composed of a set of related methods that can be used to produce one or more deliverables at software development life cycle. For example, the Structured System Analysis methodology[GS79] consists of at least five methods: data flow diagrams (DFD), data dictionary (DDic), structured English (SE), decision tables (DTab) and state transition

diagrams (STD). Normally, some methods are complementary to one another as, for example, the DTab method is to the DFD method. Others methods address similar issues in different ways as DTab to STD, or DDic to DFD. Accordingly, multiple views for a specification environment should represent the diverse aspects of the environment and allow a particular subpart of the environment to be examined while ignoring other subparts.

In Metaview, different methods are modeled by using a metamodel called the Entity-Attribute-Relationship-Aggregate (EARA) model. A model view is intended to capture the modeling process and the operational semantics of the object types available for defining a specification environment.

Definition 4.1: A *model view* is a view that contains entity, relationship, and aggregate types used to describe specification information and operations that manipulate specification information associated with these object types.

This definition refines the definition given in Definition 1.3 in the context of the EARA metamodel. The main focus of the thesis is to examine the model views and their impact on the data and control integration. For this purpose, a two-level hierarchy of model views that corresponds to different levels of semantic modeling is introduced as depicted in Figure 4.1.

In this hierarchy, two kinds of views are identified:

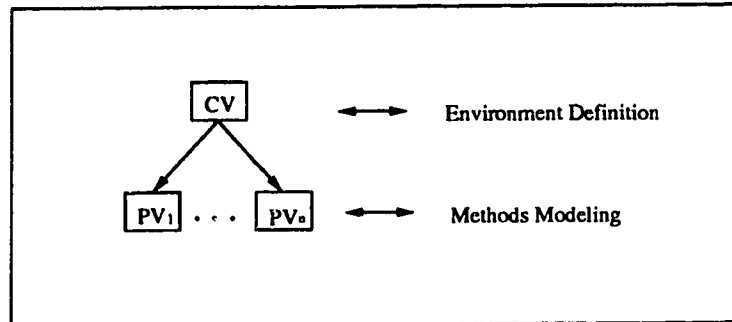


Figure 4.1: Model View Hierarchy for Semantic Modeling

- Primitive View (PV)
- Composite View (CV)

The key aspect of this hierarchy is that a specification environment can be described as a complex model view that has both an interface and an implementation. The interface specifies a protocol for the environment to communicate with other environments. The information involved includes the description of object types and services that are exported and imported by the environment and events or messages that are defined in the environment. The implementation part gives detailed definitions of the object types, associated constraints, and the operations that manipulate these object types.

This two-level modeling process is reflected in the view hierarchy as two levels of view definitions. A specification environment is represented by a composite view, which contains a collection of primitive views and operations defined on them. Primitive views encompass basic EDL/ECL statements that define the concepts and relationships needed for modeling a software engineering method, and the constraints that govern the integrity of the specification information produced by using the modeled method. Operations at this level are usually performed on the attributes associated with instances of types defined by the EARA model.

An object type may be defined differently in different primitive views. Multiple representations are therefore possible by merging all the lower level primitive views to form a composite view. Through such merging, a composite view resolves the differences between the multiple representations and thus supports sharing common information and eliminating redundancy. Operations at the level of a composite view are usually invoked implicitly and mainly for the purpose of consistency and completeness checking. In essence, merging primitive views gives us a composite view; packaging a composite view with other environment information (i.e. graphical descriptions) gives us a complete environment definition.

In order to represent ~~the~~ views, the current syntax of the EDL/ECL is extended to include the definitions of primitive views. The next section examines this issue.

4.2 Definition of Primitive Views

A *primitive view* is defined as a subschema of a specification schema. Each primitive view defines and encapsulates a set of entity types, relationship types and aggregate types, along with their associated constraints.

A primitive view may constitute a complete specification schema by itself. In this case, all the object types and associated constraints are included in a single primitive view. This is useful when an environment definer intends to use primitive views to model different ways a specification environment is defined. The DFD/CFD environment used to illustrate the definition of a composite view in Section 4.4 belongs to this category. A primitive view may also be used to capture a particular aspect of a specification environment. In this case, an environment is usually modeled by several primitive views, and the specification schema is composed of logical units formed by the union of all the primitive views. Each such primitive view can support some aspect of a software engineering method. As an example, consider the two decomposition approaches as shown in Figure 4.2¹. Either of these two styles might be used for a specification environment. However, there are some differences. Demarco's style[DeM79] of decomposition does not include the sources or destinations for boundary data flows. Gane and Sarson's style[GS79] requires the full reproduction of the boundary process and terminator entities associated with the components in an aggregate. In the current EDL implementation, if both of these styles need to be supported, two separate environments have to be generated from two sets of independent environment definitions. With the view approach, they can be modeled in two different primitive views and later merge them together into a single composite view so that data sharing is possible.

Another example of how primitive views may be used is to collect all the generic types to form a generic view. This primitive view may be used later in defining other views (composite views) to allow the generic types to be inherited by other types.

¹The diagram was taken and modified from a student software engineering course project VOCOS[S⁺91] at University of Alberta.

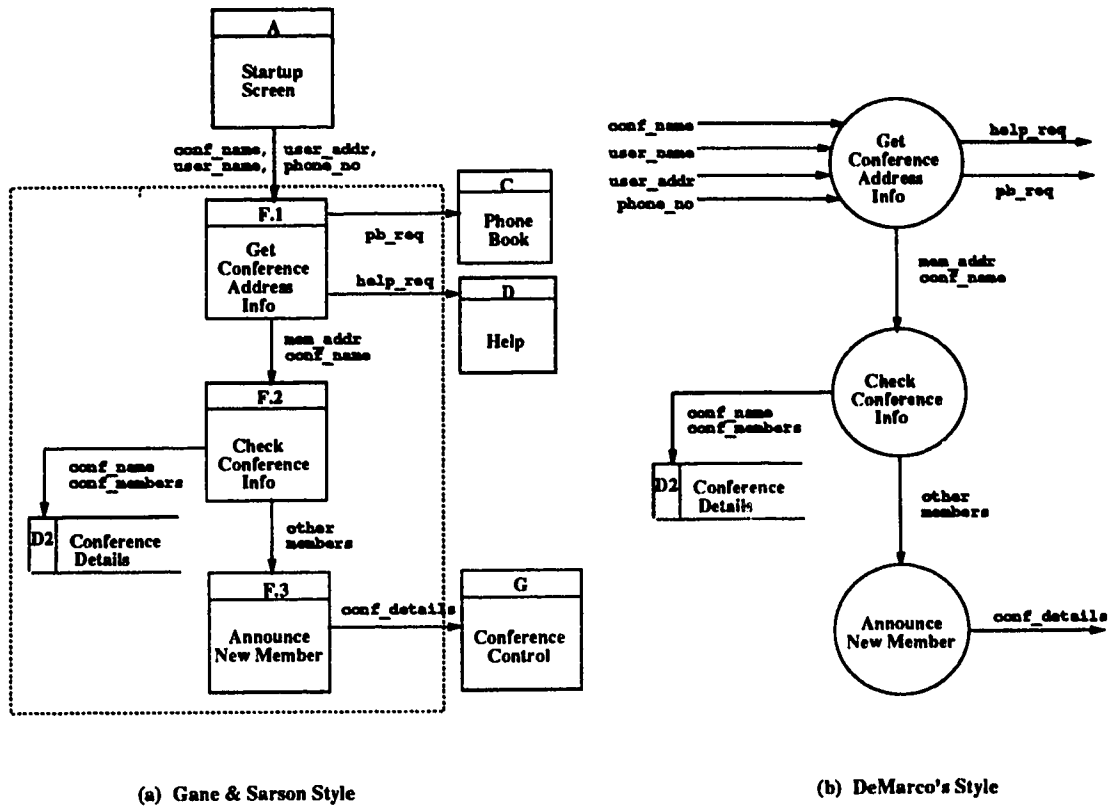


Figure 4.2: Examples of Two Decomposition Approaches

The generic form of a primitive view is given as follows:

```
P_VIEW <view_name>
    <EDL/ECL_statement>;
    {<EDL/ECL_statement>;}
END_VIEW
```

The following example illustrates the definition of an entity type **data_store** in two different primitive views of the DFD/CFD environment described in Appendix A. The first **data_store** view is defined as:

```

P_VIEW data_store_view1
  ENTITY_TYPE data_store IS_A universal
    ATTRIBUTES (ref_id: identifier,
                form: string(1..30),
                access_type: (Read_Only, Read_Write),
                number_of_copy: integer,
                primary_key: identifier);

  CONSTRAINT store_must_store IS
    --- Every data_store entity must be accompanied
    --- by a list of the data it stores.
    OBJECTS
      ds := (data_store);
    MUST_HAVE
      stores: (ds = *.store_name);
END;

  CONSTRAINT max_stores_with_data IS
    --- Each stored_data entity may participate in
    --- one and only one stores relationship.
    OBJECTS
      st := (stores);
      d := (store_data: * IN st.data);
    MUST_HAVE
      LACK_OF (stores: d IN *.data
              and * /= st);
END;
END_VIEW

```

Here, **universal** is a generic type that has several intrinsic attributes such as the **description** attribute. It is a 'free' type in a sense that it is not bound to type definitions within the scope of the **data_store_view₁** definition. The definition of **data_store_view₁** encompasses the definition of an entity type named **data_store** which inherits all the attributes of the generic type **universal** and has several attributes of its own: **ref_id**, **form**, **access_type**, **number_of_copy**, and **primary_key**. Constraints applied to this entity type dictate that a data store must store at least one type of data, and each data that needs to be stored can be stored in exactly one data store. The

'free' types, such as `universal` and `stores` will be resolved when `data_store_view1` is merged with other primitive views to form a composite view.

A second way of defining the entity type `data_store` is as follows:

```
P_VIEW data_store_view2
  ENTITY_TYPE data_store IS_A universal
    ATTRIBUTES (form: text,
                access_type: (Read_Only, Read_Write),
                index: identifier);

  CONSTRAINT form_is_mandatory IS
    --- Every data_store entity must be assigned a value
    --- for the form attribute.
    OBJECTS
      ds := (data_store);
    SATISFY
      ds.form /= "";
END;
END_VIEW data_store_view2
```

Notice that the two views overlap in non-trivial ways. While they share common intrinsic attributes (`name`, `creation_time`, `description`, etc.) and an extrinsic attribute `access_type`, the `form` attribute is different. In the first view, it has the `string` type (a character string enclosed in single quotes with the length of minimum 1 up to 30). In the second view, it has the `text` type which represents a list of character strings with no restriction on the length. The `data_store_view1` also has attributes that are not present in `data_store_view2`, i.e. `ref_id` and `number_of_copy`. The attribute `primary_key` in `data_store_view1` is the same as the `index` in `data_store_view2`, but they are named differently. The constraints imposed on `data_store` instances are also different. The first view disallows a non-empty data-store, but the second view permits it. The `data_store_view2` can be seen as an outline view of the `data_store_view1` which provides more details such as physical implementation aspects of a data store.

Nothing extraordinary has been proposed thus far. We have simply introduced a way of grouping object types and their related constraints, and allowed such groupings

to partition an environment definition into logical units. The novel aspects of the view-based approach become apparent when these primitive views are merged to form a composite view in support for information sharing and automatic control transfer.

4.3 View Merging

Central to the view-based integration is the merging of primitive views. The purpose of merging is to allow for alternative representations of entities, relationships, or aggregates that can be manipulated in a common specification database. In this section, we first identify the common elements of two or more primitive views that should be merged. We then examine how operations can be performed on the instances whose types have more than one representation.

The merging process takes place at two levels: object type level and attribute level. At the object type level, object types of the same name², but defined in two or more primitive views are merged to form a composite view. Conflicts in the definitions of the object type must be resolved. At the attribute level, the attributes of an object type that has multiple representations are coalesced to form a merged type.

For the merging purpose, we distinguish two ways, in which object types may be defined in different primitive views:

- *Synonym Types*: If an object type defined in one primitive view is also defined in another primitive view using either the identical name or a synonym, the object type is called a synonym type.
- *Homonym Types*: If an object type defined in one primitive view also appears in another primitive view with the same name, but the object types are considered to be distinct, they are called homonym types.

Synonym types participate in the merging process when multiple primitive views

²As illustrated in the next section, it is also possible to merge object types of different names through synonym identification.

are merged in order to provide information sharing and to eliminate data redundancy. Homonym types must be excluded from merging although they share the same name. Homonym types should be explicitly identified before primitive views are merged because the merging process is based on a name matching procedure through synonym identification.

Consider the **data_store** example given in the last section, there are four ways in which attributes may be defined for **data_store** entity type in different views.

- *Identical Attributes:* Attributes with the same names are defined to have the same data types, as in the case of all the intrinsic attributes. When merged, only one copy needs to be stored in a specification database. If values for a given attribute differ in two primitive views, the inconsistency has to be resolved, unless an environment definer permits such inconsistency explicitly. An example of the identical extrinsic attribute is **access_type**.
- *Unique Attributes:* Attributes are defined only in one of the views, as exemplified by attributes **ref_id** and **number_of_copy** in **data_store_view₁**. In this case, no conflicts need to be resolved and values can be stored in the database for each such attribute.
- *Synonym Attributes:* Attributes that are given different names but represent the same information (synonyms), are treated as the first case. The attributes **index** and **primary_key** in the two views exemplify this case.
- *Homonym Attributes:* Attributes with the same names are defined to have different data types (homonyms), as exemplified by the **form** attribute. They usually represent alternative ways the object types are defined. In order to provide multiple views, these different definitions are retained in the specification database.

In summary, an object type defined in different views is represented by merging the different definitions for the same attribute so that its instances have a set of instantiated copies, each of which corresponds to a certain view definition. These instantiated copies

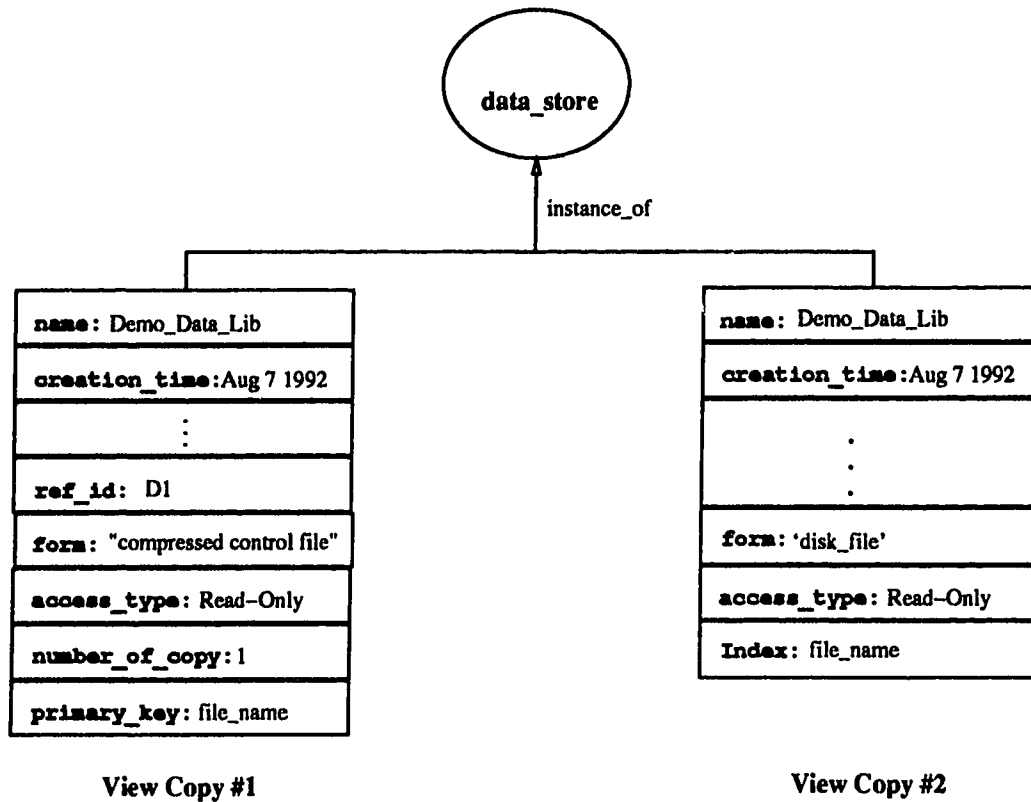


Figure 4.3: Multiple Views of Data Objects

are referred to as *view copies*. Figure 4.3 illustrates the view copies for an instance of the **data_store** type.

Constraints associated with the object types are merged by a simple union operation. As a result, stronger conditions are usually imposed on data objects as constraints from different views take effect collectively. At this stage, we assume any conflicts that may arise as a result of such a union operation are resolved manually by an environment definer.

If a specification database is constructed in such a way that multiple view copies are available for retrieval and update of database objects, the consistency of the database

objects must be maintained between different view copies, and operations on data objects must be specified in terms of the view copies to which they belong. For example, if one wishes to change the value of the `form` attribute from 'disk file' to, say, 'tutorial file', the system must be informed that this change operation is only applied to the first view of the `data_store` instance. In general, an operation, when requested, must be always accompanied by a view identifier so that the operation is performed correctly through the view. Such a view-operation pair can prevent any ambiguity that may arise during operation translations.

In Metaview, all database operations requested by a tool are performed by sending messages to the database engine through a universal interface. To support views, the format of the messages must include the view identifier. The difficulty in maintaining the consistency of such multiple view copies is that operations specified in terms of one view copy may have to be mapped to corresponding, but not necessarily identical, operations on other view copies. Section 5.1.1 will elaborate this issue.

From an implementation point of view, it is not necessary to keep a copy for each of these views in the specification database. Multiple views can be coalesced as shown in Figure 4.4.

4.4 Composite Views

In order to define a shared representation of an object in different views, some way is needed for an environment definer to describe a composite set of primitive views that collectively support a certain software engineering method. Views that provide such capabilities are called *composite views*.

A composite view description has three parts: the base views, the definition part and the operation part. Base views include a list of primitive views or other composite views that are used as a basis for building the composite view. The composite view being defined inherits all the object types defined within those base views. The definition part defines a set of merged views that are exported, along with the base views, as components

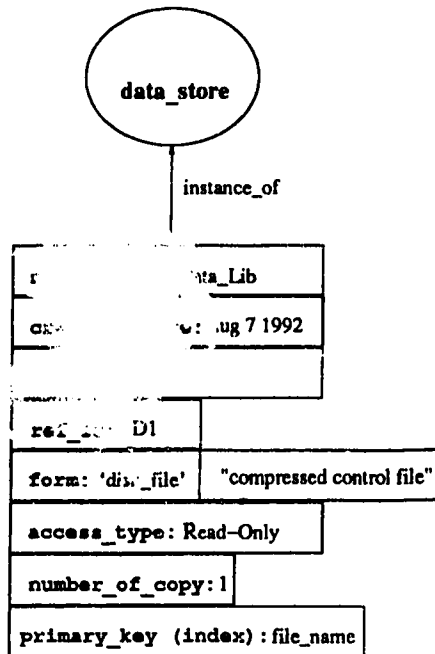


Figure 4.4: Coalescing Multiple Views

in building the environment. The operation part defines operations that should be performed to maintain the consistency of multiple view copies. In this subsection, the focus is on the first two parts. The operation part is discussed in Chapter 5. We use the structured analysis environment as an example to illustrate how a composite DFD/CFD view encompasses the primitive views that model the DFD and CFD methods.

The generic form of a composite view description is given below:

C_VIEW <c_view_name>

DEFINITION:

[**BASE_VIEW:** <view_list>;]

MERGE <view_list> | <view_defn>;

[**SYNONYM** {[<p_view_name>.<identifier>, [<p_view_name>].<identifier>}}]

[**HOMONYM** {[<p_view_name>.<identifier>, <identifier>}}];

OPERATION:

{<event> → **CONTEXT:** <view_list>

ACTION: <op-spec>;}
END_VIEW <c-view_name>

The **BASE_VIEW** clause allows other primitive views or composite views to participate in configuring a new composite view, which is important for environment evolution and reuse of view descriptions that are currently available. Views included in this clause are not involved in view merging, but are used to resolve the type names that are either inherited or used by the types in the primitive views that are to be merged by the **MERGE** clause. <view_list> is simply a list of primitive or composite view names, while <view_defn> can be any other primitive or composite view descriptions. The **SYNONYM** clause is useful when two object types or two attributes for the same object type defined in two primitive views use different names but are actually meant to be the same. The **SYNONYM** clause matches the type names or attribute names in a primitive view with their counterparts in other primitive views that are merged. The <identifier> in **SYNONYM** clause can be either a <type_name> or <type_name>.<attribute_name>. <p-view_name> is required as a prefix to <identifier> only when a naming ambiguity exists. The **SYNONYM** clauses satisfy the transitivity property. The **HOMONYM** clause is used to distinguish homonym types so that they are excluded from the merging process. The <identifier> in **HOMONYM** clause can be only a <type_name>.

Composite views have two advantages. First, a composite view provides the encapsulation mechanism so that environment can be built from component primitive views whose details are hidden. Second, a composite view allows the operations associated with primitive views to be defined. The dynamic aspects of a specification environment is thereby captured at the environment definition time.

In the operation part, an event-driven context-based rule format is chosen for specifying actions that map consistency-preserving operations from one set of views to another, and also for defining user-defined environment-specific operations. A more detailed definition of this rule format will be given in Section 5.1.3.

The two-view structured analysis environment given in Section 2.1 is now examined to see how the DFD/CFD composite view is configured syntactically.

First, three base views are defined in Appendix B.1. The **generic.view** consists of two abstract entity types: **universal** and **data_object**. The **value_type.view** collects two user-defined data types: **time_per_unit** and **ref_type**. These two views are not included in the definition of the primitive views for the DFD or CFD method because they can be re-used by other environment definitions.

The third base view **dfd_decomposition.view** is defined separately for clarity. It is assumed that two views for the structured analysis environment use the same decomposition scheme for the **process** entity, although CFD has its own decomposition scheme for the control specification.

The complete definitions of the two primitive views for the DFD and CFD methods are given in Appendix B.2 and B.3. The two methods share some common object types, such as **process**, **data_store** and **terminator**. Other object types reflect different perspectives of viewing a system requirement specification. As a result, a merged view is required. First, a control flow in CFD is treated as a special kind of data flow. It is natural to merge the two so that possible redundancy is eliminated. Similarly, a control specification (**ctrl_spec**) can be seen as a process entity that deals with only control flows (**ctrl_flow**) rather than data flows. They may differ in their graphical representations. However, that issue is out of the scope of this thesis.

Secondly, the State Transition Diagram (STD) is incorporated as a part of CFD method. In STD, **state** and **event** are unique entity types that are not present in the 'pure' DFD method. An **action** object can be seen as a name of process entity. The STD is defined as an aggregate that represents an expanded control specification.

Now that the primitive views needed to model the integrated DFD/CFD environment are identified, it is relatively easy to build the DFD/CFD composite view based on the syntax introduced earlier in this section and the view merging semantics discussed in the last section.

C_VIEW dfd/cfd_view

DEFINITION:

BASE_VIEW: dfd_decomposition_view,
 generic_view,
 value_type_view;

MERGE V_{dfd} , V_{cfd} ;

SYNONYM (ctrl_flow, data_flow)
 (ctrl_spec, process)
 (action, process)
 (data_store.index, data_store.primary_key);

OPERATION:

insert_data_flow_event(flow_type) →

CONTEXT: V_{dfd}

ACTION: IF flow_type = 'ctrl_signal' THEN
 SEND insert_ctrl_flow_event TO V_{cfd} ,
 Insert_Entity(self, data_flow),
 self.flow_type := flow_type,
 CHECK(data_must_be_sent);

insert_ctrl_flow_event(flow_name) →

CONTEXT: V_{cfd}

ACTION: Insert_Entity(self, ctrl_flow),
 SEND insert_data_flow_event('ctrl_signal') TO V_{dfd} ,
 CHECK(data_must_flow);

⋮

END_VIEW dfd/cfd_view

There are two things that require further explanation in this definition. First, the definition of a composite view allows abstract types to be merged with types defined in other primitive views, as in the case of the relationship type **access**, which is defined as an abstract supertype in V_{dfd} , but as an instantiable type in another primitive view V_{cfd} . The advantage is to allow for indirect instantiation of a supertype, which is normally prevented in the EARA model of the Metaview system.

Secondly, we have also given two examples of operation specifications for complete-

ness. Although the operational part of a composite view will not be discussed in detail until the next chapter, it is necessary to examine the issue briefly. The above two operation rules specify the actions that should be taken when a data flow instance is being created under either V_{dfd} or V_{cfd} . The purpose of these two operations is to maintain the consistency across different view copies. User-defined operations are also possible. For example, in response to a user's request to clear the entire data flow diagram, an environment definer may wish to define an operation expressed in the following rule:

```
clear_dfd_event() ←
    CONTEXT:  $V_{dfd}, V_{cfd}$ 
    ACTION:  Delete_entity(All),
             Delete_Relationship(All),
             Delete_Aggregate(All);
```

4.5 View Merging Revisited: the Algorithm

The problem of view merging is similar to that of schema integration in conceptual database design[BLN86], however, there is a major difference. In schema integration, several local schemas are restructured and merged by determining the correspondences among concepts and resolving possible conflicts. An integrated global conceptual schema is usually produced as an output. The view merging proposed in this thesis only deals with the problem of how to generate a data object in a specification database based on different definitions of the object type given in various environment description views (primitive views). Figure 4.5 shows that the view merging process is carried out in the opposite direction as schema integration.

When a specification environment is defined through simple EDL statements, the EDL compiler translates the environment definitions into PROLOG facts called environment tables that are manipulated by the database engine[Wil89] to store specification information. Such a translation is a direct process because each object type is defined only once.

There are two difficulties when multiple views are introduced in defining a speci-

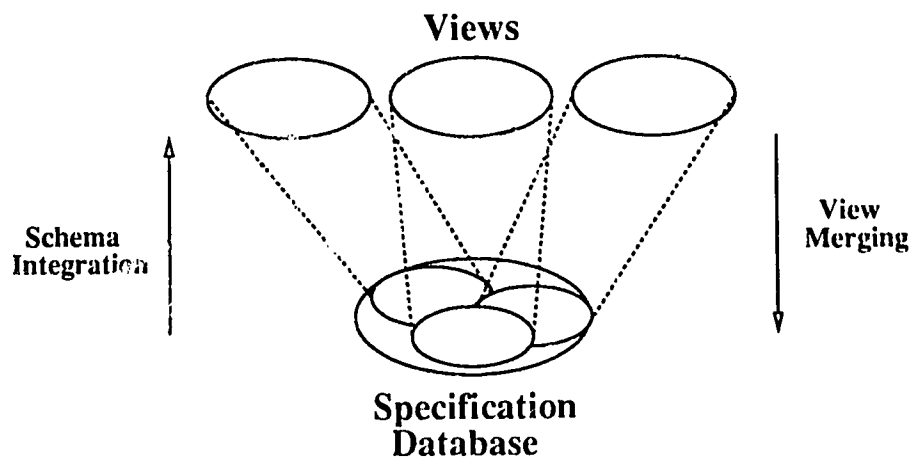


Figure 4.5: View Merging Vs. Schema Integration

fication environment. First, the definition of an object type may be widely dispersed across many primitive views. In order for a database engine to construct and maintain instances of the object type, various definitions of a given object type have to be coalesced into a merged type. This involves merging between multiple primitive views to form a composite view. A composite view may also participate in such a merging process through primitive views defined within its scope. Conflicts may arise when the merging process attempts to resolve the differences between primitive views. For example, the same relationship may have different participants in two primitive views, and the same aggregate object may have different components, etc. Secondly, the Metaview system has to maintain two kinds of associations: for each object type, a set of associated primitive views that define it, and for each attribute, a subset of those primitive views in which that attribute appears in the definition of an object type. Such associations are needed so that operations specified with respect to views can be translated correctly to operations on basic data objects of the merged type. Two tables are created to record these two kinds of associations: **TYPE** and **ATTR**. **TYPE** contains lists of the form: (T, V_1, V_2, \dots) , where T is an object type, and the V_i are all the primitive views that define T . The table **ATTR** contains all the lists of the form $(T, A, V'_1, V'_2, \dots)$, where A is

an attribute name and the V_i' are the subset of V_i .

Functions/Procedures that are used to describe the algorithms are given in Table 4.1. It is assumed that these functions are pre-defined, and available for use.

The merging process is described in the following algorithm.

merge(V_1, V_2, \dots, V_n)

--- This procedure returns a composite view represented as a set of merged data structures for each object type and two lists, TYPE and ATTR, which describe the associations between object types, their attributes and their defining primitive views.

V_i : view_name; ($i = 1..n$)

T : object type;

TYPE, ATTR: list;

{

FOR $i = 1$ TO n

--- If V_i is a composite view, the merge algorithm is applied recursively to a list of primitive views that appear in the MERGE clause of V_i

IF *composed_of*(V_i) \neq null THEN

$V_i := \text{merge}(\text{composed_of}(V_i));$

END_FOR

collect_view_defn(V_1, V_2, \dots, V_n); --- construct the list TYPE

name_matching(T); --- identify synonyms and homonyms,
the list TYPE is modified accordingly.

FOR each $T \in \bigcup_{i=1}^n V_i$

CASE T

entity_type:

coalesce(T); --- construct the list ATTR

relationship_type:

merge_relationship(T);

aggregate_type:

merge_aggregate(T);

END_CASE

END_FOR

}

Functions/Procedures	Description
<i>composed_of(V)</i>	returns a list of primitive views that are involved in <i>V</i> 's MERGE clause. If <i>V</i> is a primitive view, the function returns null.
<i>defined(T, V)</i>	returns true if object type <i>T</i> is defined in view <i>V</i> .
<i>synonym_of(T)</i>	returns <i>T</i> 's synonym given in the SYNONYM clause of a composite view definition.
<i>homonym_of(T)</i>	returns <i>T</i> 's homonym given in the HOMONYM clause of a composite view definition.
<i>homonym_in(T)</i>	returns the view name in which <i>T</i> is identified to be another object type's homonym type.
<i>new_name()</i>	returns a unique identifier.
<i>append(l, V)</i>	appends <i>V</i> to the list <i>l</i> .
<i>remove(l, TYPE ATTR)</i>	removes list <i>l</i> from either TYPE or ATTR.
<i>add(l, TYPE ATTR)</i>	adds list <i>l</i> to either TYPE or ATTR.
<i>delete(l, V)</i>	deletes <i>V</i> from the list <i>l</i> .
<i>get_list(T)</i>	returns the list (<i>T</i> , <i>V</i> ₁ , <i>V</i> ₂ , ...) in TYPE.
<i>in_list(V, l)</i>	returns true if <i>V</i> is an element of list <i>l</i> .
<i>length(l)</i>	returns the number of elements in the list <i>l</i> .
<i>type_of(A, T, V)</i>	returns the type name of the attribute <i>A</i> of type <i>T</i> in view <i>V</i> , returns null when <i>A</i> is not defined.
<i>participants(T, V)</i>	returns the list of participants of the relationship type <i>T</i> defined in view <i>V</i> .
<i>combine_component_list(T)</i>	returns a list that contains all the component types defined for the aggregate type <i>T</i> in different primitive views.
<i>resolve_conflicts(T)</i>	if merging fails for a relationship type <i>T</i> , manual intervention is required.

Table 4.1: Functions/Procedures For Merging Process

We now elaborate the functions that appeared in this algorithm:

(Collect View Definitions):

```

collect_view_defn( $V_1, V_2, \dots V_n$ )
i: integer;
ltemp: list;
{
  FOR each T
    --- For each object type T, add a list (T,  $V_1, V_2, \dots$ ) to TYPE,
        where  $V_i$  are all the primitive views that define T.
    ltemp = (T); --- initialize the list ltemp
    FOR i = 1 TO n
      IF defined(T,  $V_i$ ) THEN
        append(ltemp,  $V_i$ );
    END_FOR
    add(ltemp, TYPE);
  END_FOR
}
```

(Establish Naming Correspondences):

```

name_matching(T)
--- In this function, synonyms and homonyms are identified and the list
    TYPE built in collect_view_defn() is modified accordingly.
T': object type;
lT, lT', ltemp: list;
{
  T' := synonym_of(T);
  IF T' ≠ null THEN
    --- Different identifiers are used to refer to the object types that are
        mergeable. A unique identifier is given for them, and entries
        in TYPE that involve T and T' are deleted from TYPE, a new
        entry that combines these two lists is added to TYPE.
    {
      lT := get_list(T);
      lT' := get_list(T');
      ltemp := (new_name()); --- initialize the list ltemp with a unique name.
      FOR each V ∈  $l_T \cup l_{T'}$ 
```

```

    --- append all the view names that either define  $T$  or  $T'$  to the list  $l_{temp}$ 
    IF NOT in_list( $V, l_{temp}$ ) THEN
        append( $l_{temp}, V$ );
    END_FOR
    add( $l_{temp}, TYPE$ );
    remove( $l_T, TYPE$ );
    remove( $l_{T'}, TYPE$ ); ---  $l_T$  and  $l_{T'}$  are replaced by the new list.
}

 $T' := homonym\_of(T)$ ;
IF  $T' \neq \text{null}$  THEN
    --- The same identifier is used to refer to two different object types,
    the list in  $TYPE$  that involves  $T$  should be split into two lists.
    {
         $l_{temp} = delete(get\_list(T), homonym\_in(T))$ ;
        remove( $get\_list(T), TYPE$ );
        add( $l_{temp}, TYPE$ );
        add( $(T', homonym\_in(T)), TYPE$ );
    }
}

```

(Merge Entities):

Objects of entity types are merged by simply coalescing all the attributes defined in different views. The *coalesce*(T) procedure is also used in merging relationship types and aggregate types.

coalesce(T)

--- For each attribute A of T , visit the definition of T in each of T 's contributing primitive views. If A is defined in only one view or defined to have the same data type in multiple views, a triple (T, A, V) is added to $ATTR$. If A is defined differently in multiple views, a list that contains these view names is added to $ATTR$.

i, j : integer;

A : attribute_name;

diff_defn: Boolean; --- true if an attribute has an alternative definition.

l_T, l_{temp} : list;

```

{
     $l_T = get\_list(T)$ ;
    IF length( $l_T$ ) > 2 THEN
        --- If true,  $T$  must be defined in multiple views.

```

```

FOR each  $A \in T$ 
   $l_{temp} = (T, A)$ ; --- initialize the list that is to be added to ATTR
  FOR  $i = 1$  TO  $n$ 
    IF  $in\_list(V_i, l_T)$  THEN
      --- examine all the views that define the object type  $T$ .
      {
         $diff\_defn = false$ ;
        FOR  $j = 1$  TO  $i - 1$ 
          IF  $in\_list(V_i, l_T)$  AND  $type\_of(A, T, V_i) \neq type\_of(A, T, V_j)$  THEN
            --- if attribute  $A$  is defined differently in view  $V_i$  than in any of
              the views that have already been appended to the list  $l_{temp}$ ,
              then  $V_i$  should also be appended to  $l_{temp}$ .
            {
               $diff\_defn = true$ ;
              BREAK;
            }
          }
        END_FOR
        IF ( $diff\_defn$  AND  $type\_of(A, T, V_i) \neq null$ ) THEN
           $append(l_{temp}, V_i)$ ;
        }
       $add(l_{temp}, TYPE)$ ; --- add the constructed list to ATTR.
    END_FOR
  END_FOR
}

```

(Merge Relationships):

$merge_relationship(T)$

i, j, p, q, l_1, l_2 : integer;

l_T : list;

P_List_1, P_List_2 : list;

--- Participant lists. Suppose $P_List_1 = (P_1, P_2, \dots)$ and

$P_List_2 = (P'_1, P'_2, \dots)$

```

{
   $l_T = get\_list(T)$ ;
  IF  $length(l_T) > 2$  THEN
    FOR each  $i \neq j$  AND  $in\_list(V_i, l_T)$  AND  $in\_list(V_j, l_T)$ 
       $P\_List_1 = participants(T, V_i)$ ;
       $P\_List_2 = participants(T, V_j)$ ;
    }
  }

```

```

     $l_1 := \text{length}(P\_List_1);$ 
     $l_2 := \text{length}(P\_List_2);$ 
    FOR  $p = 1$  TO  $l_1$ 
        FOR  $q = 1$  TO  $l_2$ 
            IF  $P_p = P'_q$  OR  $P_p = \text{synonym\_of}(P'_q)$  THEN
                {
                     $\text{delete}(P\_List_1, P_p);$ 
                     $\text{delete}(P\_List_2, P'_q);$ 
                }
            END_FOR
        END_FOR
    END_FOR
    IF  $\text{length}(P\_List_1) = 0$  OR  $\text{length}(P\_List_2) = 0$  THEN
        --- one participant list is a sublist of another, and all participating
        --- entities paired up are identical, or can be merged through
        --- identifying synonyms.
         $\text{coalesce}(T);$ 
    ELSE
        --- the two definitions of the relationship type  $T$  are not considered
        --- to be mergeable, and an environment definer is called up to help
        --- resolve the difference manually.
         $\text{resolve\_conflicts}(T);$ 
    }

```

(Merge Aggregates):

```

 $\text{merge\_aggregate}(T)$ 
    ---  $T$  is an aggregate type.
     $l_T$ : list;
    {
         $l_T = \text{get\_list}(T);$ 
        IF  $\text{length}(l_T) > 2$  THEN
             $\text{coalesce}(T);$ 
             $\text{combine\_component\_list}(T);$ 
            --- allow the merged aggregate type to contain all the component
            --- types defined in different views.
        }
    }

```

The table 4.2 shows part of the two lists produced when applying this algorithm to the DFD/CFD environment. In this table, `process_spec` is the merged type for

TYPE	ATTR
(process_spec, V_{dfd} , V_{cfd})	(process_spec, identification, V_{dfd}) (process_spec, function_description, V_{dfd}) (process_spec, physical_location, V_{dfd})
(data_store, V_{dfd} , V_{cfd})	(data_store, ref_id, V_{dfd}) (data_store, form, V_{dfd} , V_{cfd}) (data_store, access_type, V_{dfd} , V_{cfd}) (data_store, number_of_copy, V_{dfd}) (data_store, primary_key, V_{dfd})
(data_ctrl_flow, V_{dfd} , V_{cfd})	(data_ctrl_flow, flow_type, V_{dfd})
(terminator, V_{dfd})	
(sends, V_{dfd})	(sends, frequency, V_{dfd})
(stores, V_{dfd})	
(access, V_{dfd} , V_{cfd})	(access, access_type, V_{cfd})
(changes, V_{dfd})	
(reads, V_{dfd})	
(event, V_{cfd})	(event, event_type, V_{cfd})
(state, V_{cfd})	(state, state_type, V_{cfd})
...	...

Table 4.2: TYPE and ATTR for DFD/CFD Environment

`process`, `ctrl_spec` and `action`, and `data_ctrl_flow` is the merged type for `data_flow` and `ctrl_flow`. These two lists provide only the static aspect of the data representations. The dynamic links between different object types that are defined in multiple views will be discussed in Chapter 5.

4.6 Comparison with Other Views

We compare our views with two related views that are common in database design: relational views and object-oriented views. The discussion is based on the fact that the multiple view support in our case is built on the given meta data model (EARA model), which is already well defined. We do not claim our view mechanism is necessarily better

than these two approaches in all aspects. The purpose is to point out the similarities and differences between our approach and others.

4.6.1 Relational Views

Relational views are derived from relations (tuples composed of simple data types) by using relational operators (projection, selection, join and Cartesian product). Our views are constructed by merging object types that are possibly much more complex. Relational views are basically passive views, while ours are more constructive in a sense that we define primitive views first and the way they are merged guides the construction of the database.

Relational views are usually virtual views. Materialized relational views are also present in some relational databases. However, the purpose of materialization is mainly for retrieval efficiency. With virtual relational views, operations performed on a relational view must be translated into operations on the base relations on which the views are defined. The update through views often causes the problem of semantic ambiguity, which is impossible to be resolved properly without user's intervention. For example, when a request that asks for the deletion of a player from a school football team is raised from a football coach's view, it could actually mean to simply change the attribute "team_membership" from "yes" to "no" when translated to base relations containing the student's personal data. However, if the student is removed from the team because of death, his record should be erased. Semantic update ambiguity is not present in our case, because our views are the instantiation of merged object types that represent the multi-faceted nature of the data objects. Operation translations are deterministic and unambiguous.

Relational views do not include any operational functions of data management. The database management system must take the responsibility of consistency checking and other explicit management services. Our views encapsulate such operations such that the database consistency can be preserved automatically.

4.6.2 Object-Oriented Views

One major difference between views and the object-oriented paradigm lies with the multiple representations for an object. At the first glance, mechanisms that support multiple inheritance seem to be very similar to what is needed to support multiple views, insofar as both allow for merging a collection of independently defined types into a new composite type. However, when an object inherits data types and functions from several supertypes, all data types and functions are combined into a common pool. Conflicts must be resolved before joining the pool. After conflicts are resolved, data types are still of a single representation, and only one set of functions are performed on these data types. When an object is said to have multiple views, support for integrated, but separately materialized view copies must be provided. The object appears to be multi-faceted. Operations are performed with respect to one of these view copies, and explicit mappings may be necessary to invoke operations on other view copies in order to maintain the consistency among them.

In the following discussion, we use the FUGUE object views [HIZ88, Hiei90] as an example in comparing our views with the object-oriented views. FUGUE is an object-oriented model built on three fundamental concepts: objects, functions, and types. The basic execution paradigm is function application. Objects are the things that functions are applied to. They are the inputs and outputs of functions. Functions are applied to objects to yield their attributes or related objects, or to test constraints on the objects, or to perform operations on objects. Associated with each function is a procedure that when invoked with input arguments produces the associated output arguments. Objects are organized into types by the functions that can be legally applied to them, i.e., the functions that specify their structure, properties, and behavior. In FUGUE, a view is a *context* in which functions are applied to objects. A view defines a set of objects and binds each to a type that determines the functions that can be applied to them or that yield them in that view. A view in FUGUE hides or exposes methods as well as data. There are three ways of binding objects to specific types in a view:

- restrict the objects to a subset of instances of the specified types,

- restrict the types to a subset of the possible types of the objects, and
- restrict the functions that can be applied to objects.

An object is allowed to be viewed differently when bound to different types. Multiple views are therefore possible to have the same set of objects that differ in the set of functions.

Similar to our views, a function in the FUGUE model always accesses an object by means of a particular view to avoid possible ambiguities. The differences between our views and FUGUE views are mainly in the following three aspects:

- The ways that multiple views are achieved are different. In FUGUE, functions are used to define how objects are bound to different types. In our views, multiple views are defined statically when a specification environment is defined.
- The operation mapping mechanisms are different. In FUGUE, functions are bound to types. Objects of a specific type have their own set of functions that are only applicable to the instances of the type. In our views, mappings are defined explicitly by the EXA rules. The EXA rules have been introduced briefly in this chapter, and will be discussed in more detail in the next chapter.
- Views in FUGUE are closed. Each of the functions of types to which objects in a view are bound yields objects that are also in the view and that are bound to types in the view. Our views, however, do not impose the restriction on what objects should be included in a view. Operations performed on objects in one view may invoke other operations on objects that belong to different views.

Chapter 5

Specification of View Operations

In the last chapter, we discussed merging of primitive views to form a composite view. This is considered to be the static aspect of the view integration. In this chapter, we will discuss the dynamic part of the integration: the operation mappings. The purpose of such mappings is to assist in automatic maintenance of the consistency of a specification database. The mechanism designed for this goal is the EXA rules. After a framework for formalizing operation specifications is given, the example Structure Analysis Environment is re-examined to see how the DFD method can be integrated dynamically with the CFD method. With the EXA rule mechanism, a specification environment with multiple views is able to work in such a way that operations performed with respect to one view are automatically mapped to appropriate operations on other views. In this way, the control integration of a specification environment is achieved.

5.1 Operation Mappings

In Section 4.4, we discussed what should constitute a composite view. Little was said, however, about the operations that are needed to maintain consistency between view copies or to perform user-specified functions. Before we define this operational part of the composite view, the following two questions should be answered:

- What constitutes a specification database state that supports multiple views?
- How is the consistency of a specification database maintained when operations are performed on view copies?

Based on the answers to these two questions, the following question can then be addressed:

- How is a consistency-preserving operation mapping over different view copies specified?

This section will provide answers to these three questions.

5.1.1 The Database State

A specification database in Metaview is made up of three types of objects¹:

- instances of entity types,
- instances of relationship types, and
- instances of aggregate types.

Each of these three types of database objects contains intrinsic and possibly extrinsic attribute values. A relationship object also has a list of participants linked to each role of the relationship. An aggregate object has a list of component objects that constitute a relatively independent information module. The definition for the database state is given as follows:

Definition 5.1: A *multi-view specification database state* is a snapshot of the database at a particular point in time, characterized by the following hierarchy of values and object instances:

- a) values of simple data types, such as **integer**, **boolean**, etc.
- b) instances of structured types, such as **list**, **record**, and **text**, which contain values from a)
- c) instances of attributes of object types, which contain instances from b)

¹Objects defined with the graphical extension of the EARA model are not included.

d) view copies (instances of object types).

A database state is said to be consistent if the database satisfies all the constraints, called *semantic integrity constraints*[OV91, page 158]. When data objects are replicated, a database is in a *mutually consistent state* when all the copies of every data item have identical values[OV91, page 258].

In a specification database of the Metaview system, constraints are classified into four categories as shown in Table 3.1. Except for the environment generation constraints, which guide the implementation of the Metaview system itself, the other three types of constraints affect the consistency of the Metaview specification database. A complete list of constraints that must be satisfied by any database state can be found in [Fin92]. With the introduction of multiple views, the consistency between the view copies of a data object is particularly important when operations are performed on view copies. The next subsection will discuss how such consistency is preserved.

5.1.2 Consistency-Preserving Operation Mappings

The last subsection gave a definition of the database consistency, but did not explain how to achieve the consistency when operations are performed on database objects with multiple representations. In this subsection, the concept of *consistency-preserving operation mapping* is introduced. The intuition behind this concept is as follows. Assume an operation Op_1 is executed in the context of a view V_1 . The challenge is to insure that after the completion of Op_1 the database remains consistent with respect to some other view, say V_2 , on the database. To meet this challenge, it may be necessary to invoke another operation, Op_2 , on V_2 . If this is required, then a mapping can be established such that $M(Op_1) \equiv Op_2$. An operation in this context is possibly composed of a sequence of basic database manipulation commands.

Definition 5.2: Suppose V_1 and V_2 are two view copies of the same data object of a consistent database, and Op is an operation defined on V_1 . A mapping M is said to be a *consistency-preserving operation mapping* if it maps Op to operation $M(Op)$ on V_2 , so that when both operations $Op(V_1)$ and $[M(Op)](V_2)$ are performed, the state of the

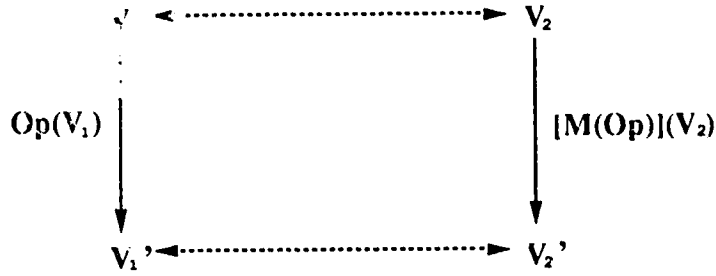


Figure 5.1: Consistency-Preserving Operation Mapping

specification database remains consistent.

This definition is similar to that given in [Gar87]. The difference is that our focus is on the consistency-preserving nature of a mapping, rather than the mapping compatibility. Figure 5.1 illustrates this concept graphically.

The definition of specification database consistency was given in the last subsection. If an operation results in an inconsistent database state, some other operations that are predefined as its mapping should be performed in order to preserve the database consistency. In order to see whether such a mapping is a consistency-preserving operation mapping, type definitions that represent alternative views of an object are first identified. This is done by matching type definitions with the same name in each view copy. For each matched type definition, attributes with the same name are examined. If they have the same type, the mapping is considered to be consistency-preserving only when the same operations are performed on the different view copies. If they have different types, the mapping should be established according to the constraints definition. The following example illustrates how to decide a consistency-preserving mapping.

In the DFD/CFD environment defined earlier, the relationship type **scores** is defined to have a list of **data_flow** objects occupying one of the two roles. Let's also assume that in another view, it requires a **set** of **data flow** objects occupying the same role. A list may contain duplicates, but a set cannot. A **Delete** operation on a set can be mapped to a **Delete_all_occurrence** operation on a list. After both operations are

performed, the consistency of the specification database is maintained. It would also be possible for operation on one view to be mapped onto nil on the other, where nil means no operation. For example, rearranging the order of the elements in a list has no counterpart on a set.

5.1.3 EXA Rules

Consistency-preserving operation mappings are represented syntactically using event-triggered context-based action (EXA) rules proposed in this subsection. Such EXA rules can also be used to specify developer-required functions following the uniform syntactic representation as illustrated at the end of Section 4.4. The idea of EXA rules was inspired by the work of U. Dayal, *et al.* on the ECA rules introduced in their HiPAC project [UDMS8, D⁺88]. In their case, ECA stands for Event, Condition, and Action. A system monitors events, when an event occurs, conditions are evaluated, and actions are taken if conditions are satisfied. ECA rules in HiPAC project provide timely response for database management functions such as integrity checking, access control, and view management.

The EXA rules proposed in this subsection are different. Our EXA rules consist of three parts: *Event*, *context*, and *Action*. The purpose of such EXA rules is to provide a mechanism for context integration in a specification environment. The basic idea is as follows: events are monitored by a multi-view specification environment. When such an event occurs, actions are taken corresponding to the context in which the event occurs.

The event can be a system supported concept, such as: transaction commit, a clock tick, a request for a read/write operation on an object; or an external stimuli, such as a mouse click or other user-specified requests. It is parameterized so that information is passed with the event name to assist in decision-making or in carrying out the actions. For the purpose of environment integration, we are mostly interested in a particular type of events: the messages that are passed between various views of an environment. The discussion below will focus on this type of event.

The context part is a list of view names that help to identify where an event occurs. It is important because the same event may trigger different actions when it occurs within different views.

The action part specifies the operational behaviors caused by an event in the specified context. When an EXA rule is fired, a system developer must be working under one of the pre-defined views. This context determines which corresponding actions should be taken. In formalizing operational behaviors required by a specification environment, we propose the following four types of actions:

- constraint-checking
- conditional branching
- message-passing
- database manipulation

Examples of a parameterized event were demonstrated in Section 4.4. In the following, it is assumed that all operations are associated with certain event names and appropriate information is passed as event parameters. The four types of actions are represented by the syntax described below:

Constraint-Checking Actions:

CHECK(<constraint_list>) [AUTOMATIC | ON_REQUEST]

Conditional-Branching Actions:

IF(<expression>) THEN <actions>
 {ELSE <actions>},
 <actions>: <expression>

Message-Passing Actions:

SEND <messages> TO <view_list>

Database-Manipulation Actions:

<db_engine_routine_name>(<parameter_list>)

As already pointed out in Section 3.3.2, the generalized specification constraints and environment generation constraints (Class I and IV in Table 3.1) are usually checked

automatically when an environment is generated or when database engine routines are executed. But how the environment specific constraints are checked may vary from environment to environment. The constraint-checking actions, which take the form of a function `CHECK`, tell the system when and what constraints should be checked. This provides an environment definer with the full control of how constraint checking is invoked to maintain the database consistency. The keyword `AUTOMATIC`, which is the default, indicates that the checking is performed automatically. System developers are informed only when a violation is detected. The keyword `ON_REQUEST` indicates that the checking is done only on developer's explicit request when the developer is in a process of developing software specifications in a specification environment.

The conditional branching provides a programming language-like decision-making capability. The syntax follows the `IF-THEN-ELSE` style. The `<actions>` following `THEN` and `ELSE` can be any of the four types of actions. The `<expression>` is a logical expression commonly seen in any programming languages.

The message-passing actions are useful for mapping operations on one view to operations on another. The message-passing approach is chosen as the vehicle for defining operation mappings. This allows operations to be associated with views that carry out the execution, not views that invoke them. The `<messages>` is a list of messages separated by commas. And `<view_list>` is a list of views that receive the messages. Messages passed between views are treated as triggering events. When such a message is received within a composite view, its associated operations are performed.

The last type of actions are database manipulation actions. They are operations performed by the database engine routines [Wil89]. We distinguish a database operation from a request for accessing the database. The former is expressed in the action part of an EXA rule, while the latter is considered to be a message which is sent to related views through a message-passing action for the execution. There are five types of database operations: insert, delete, change and retrieve, as well as database initialization and close operations. Examples are `insert_entity(name, type)`, `insert_relationship(name, type)`, etc. Because database engine routines are not intended for the direct use even by an

Name of Actions	Parameters
insert_aggregate	aggregate_name, aggregate_type
insert_entity	entity_name, entity_type
insert_relationship	relationship_name, relationship_type
delete_aggregate	aggregate_name
delete_entity	entity_name
delete_relationship	relationship_name
change_aggregate_attribute	aggregate_name, attribute_name, attribute_value
change_entity_attribute	entity_name, attribute_name, attribute_value
change_relationship_attribute	relationship_name, attribute_name, attribute_value
retrieve_aggregates	aggregate_type, condition
retrieve_entities	entity_type, condition
retrieve_relationships	relationship_type, condition
retrieve_aggregate_attribute	aggregate_name, attribute_name
retrieve_entity_attribute	entity_name, attribute_name
retrieve_relationship_attribute	relationship_name, attribute_name

Table 5.1: Database Manipulation Actions

environment definer, we have adopted a simplified notation for database manipulation actions. Table 5.1 gives a list of functions that can be used to specify database manipulation actions. Note that we have extended the retrieve operation to allow selective retrieval based on a given logical condition. Also, for convenience, we use assignment statements to replace the `change_attribute` functions.

There are two reasons why the specification of operations is not relegated to primitive views. First, an important goal was to keep the Metaview's EARA model intact while developing a mechanism to associate operational semantics with the meta model. The primitive views encompass all the declarative semantics of the model through EDL/ECL statements, so it is not a good idea to insert the operation specifications at this level. Secondly, the granularity of a primitive view is too fine for effective function sharing among object types. A composite view is capable of providing both detail hiding and polymorphism which are required when specifying operations.

As an example, a relationship type named **access** is examined.

In Appendix B, the **access** relationship is defined differently in two primitive views, V_{dfd} and V_{cfd} . In V_{dfd} , it is a generic type, and cannot be instantiated. Two relationship types, **changes** and **reads**, are defined as its subtypes. In V_{cfd} , it is an object type that can be instantiated, and has an additional attribute **access_type** which takes a value of either 'detects' or 'modifies'. In this case, these two subtypes are not needed. In keeping both views, an operation mapping is needed to invoke corresponding operations in V_{dfd} when the **access_type** is changed in V_{cfd} from 'detects' to 'modifies' or vice versa. The rule for this operation mapping is defined as follows:

```

access_type_event(new_access_type) —
CONTEXT:  $V_{cfd}$ 
ACTION: self.access_type := new_access_type,
        IF (new_access_type = 'detects') THEN
            {
                changes_obj := retrieve_relationships(changes,
                    "*.p_name = self.process_name AND
                    *.data = self.data"),
                SEND (insert_reads_event(self.process_name, self.data),
                    delete_changes_event(changes_obj))
                To  $V_{dfd}$ 
            }
        ELSE
            {
                reads_obj := retrieve_relationships(reads,
                    "*.p_name = self.process_name AND
                    *.data = self.data"),
                SEND (insert_changes_event(self.process_name, self.data),
                    delete_reads_event(reads_obj))
                To  $V_{dfd}$ ;
            }

```

In the rule, the first action statement replaces the access type of the **access** instance with the **new_access_type**. After the access type is replaced, corresponding messages are sent to V_{dfd} , where subtypes **changes** and **reads** are defined, to create new instances

of the relationship types **changes** and **reads**, while deleting the old ones. A system variable **self** is used to identify the data object on which an operation is performed. In this rule, it is assigned the instance name of the relationship type access, whose value for the attribute **access_type** is changed. It is assumed that when a message is sent from one view to another, the value of the variable **self** is propagated to the rules that are invoked by the message sent. An execution model that invokes appropriate rules based on the occurrence of events (e.g., message sending) is not discussed in this thesis, and remains a topic for future research.

5.2 Defining EXA Rules: an Example

In the last subsection, a framework of formalizing operation specifications was suggested. This section examines some of the events and their associated actions within two different view contexts in the DFD/CFD environment. A more complete set of EXA rules defined for the DFD/CFD environment is provided in Appendix B.4.

5.2.1 Assumptions

The following assumptions are made when defining the EXA rules for the DFD/CFD Environment.

1. Only events that change the specification database states are considered. Other events, such as system signals, are not discussed in this section.
2. Actions that map operations between distinct views are our major concern. Thus, most of constraint-checking operations are omitted in the following operation specifications. But they can be easily added to the environment definition when the environment is defined.
3. Operations on data objects with only a single representation are propagated to all the view copies automatically. For example, when a **terminator** object is created

within V_{dfd} , another view, V_{efd} , should be informed of the creation of this object immediately.

4. Change operations are only applicable to attribute values of an object type. When the participant name lists of a relationship object, or the component name list of an aggregate object are changed, they will be treated the same as deleting the old relationship or aggregate object and inserting a new one.
5. The following operations are performed automatically by the Metaview database engine as a result of the meta-level constraint checking. They need not to be specified explicitly in EXA rules:
 - When an entity object is deleted, all associated relationship objects in which the entity object is a participant are also deleted.
 - When an object is inserted into the database, its name is checked so that no object that is of the same type belonging to the same aggregate has the identical name. The attributes of the inserted object are also checked to see if their values have appropriate types.
 - When a relationship object is inserted to the database, its participating entities are checked to see if they belong to the same aggregate object, and do not replicate participants of another relationship object within the same aggregate.
 - When an aggregate object is inserted into the database, its component objects are checked to see if they have the required types.
 - When an attribute of an object is changed in the context of one view, and the attribute has a single representation in several views, the change is propagated to the other views automatically.

5.2.2 Events in DFD/CFD Environment

Considering operations defined by the Metaview database engine², the following cases need to be taken into account for consistency-preserving mappings,

- Insert a data object. If the object to be inserted is an instance of a merged type, all the view copies will be created. Only those attributes appearing in the specified view copy are assigned values.
- Delete a data object. All the view copies of this data object are deleted unless some explicitly defined constraint dictates that only the related view copies be deleted.
- Change an attribute value of a data object. The old attribute value in the specified view copy is replaced by the new one, and operations are performed on other view copies to maintain the database consistency.

Retrieving data objects or attributes of a data object does not change a database state. Therefore, the consistency is preserved and no operation mappings are necessary. When such operations are performed, only relevant information under a particular view is presented as the results of the retrieval. In this way, the information hiding capability for views is provided.

Table 5.2 lists the events that are applicable in the DFD/CFD environment.

5.2.3 Operation Specifications in DFD/CFD Environment

This subsection provides three example rules defined for the DFD/CFD environment. These rules correspond to the three types of events listed in the last subsection, i.e. inserting a data object, deleting a data object, and changing an attribute of a data object.

²More complete descriptions of the generic database operations can be found in [Wil89].

Event_Name	Parameters	Context
insert_data_store_event	form_value	V_{dfd}, V_{cfd}
insert_data_flow_event	flow_type	V_{dfd}
insert_ctrl_flow_event		V_{cfd}
insert_changes_event	process_name, data_name	V_{dfd}
insert_reads_event	process_name, data_name	V_{dfd}
insert_access_event	access_type	V_{cfd}
insert_sends_event	data_sent, from, to, freq_value	V_{dfd}
insert_in_out_event	input_flow, process_cspec, output_flow	V_{cfd}
insert_transition_event	from, event_name, actions, to	V_{cfd}
delete_data_flow_event		V_{dfd}
delete_ctrl_flow_event		V_{cfd}
delete_changes_event	changes_object	V_{dfd}
delete_reads_event	reads_object	V_{dfd}
delete_access_event	access_object	V_{cfd}
delete_sends_event		V_{dfd}
delete_in_out_event	in_out_object	V_{cfd}
delete_transition_event	transition_object	V_{cfd}
change_form	form_value	V_{dfd}, V_{cfd}
change_access_type	new_access_type	V_{cfd}
change_flow_type	flow_type	V_{dfd}

Table 5.2: Events in DFD/CFD Environment

Example 1: The first rule states that when a data store object is inserted in the V_{dfd} view, the system should send an event to V_{cfd} view so that another copy of the data store is created in V_{cfd} with the form attribute set to text type.

```

insert_data_store_event(form_value) →
    CONTEXT:  $V_{dfd}$ 
    ACTION:  Insert_Entity(self, data_store),
             self.form := form_value,
             SEND insert_data_store_event(string_to_text(form_value))
             TO  $V_{cfd}$ ;

```

The corresponding rule in V_{cfd} that is invoked based on the message passing action

is:

```
insert_data_store_event(form_value) →  
  CONTEXT:  $V_{cfd}$   
  ACTION: Insert_Entity(self, data_store),  
          self.form := form_value,  
          SEND insert_data_store_event(text_to_string(form_value))  
          To  $V_{dfd}$ ;
```

There is one potential problem in implementing the operation mapping. When a data store is created in one of the views, a message is sent to another view. This could form a cycle, in which events are sent back and forth between views. The solution is as follows: when a view receives a message, it first checks if it is the result of an operation mapping (i.e., if the message is sent by another primitive view). In this case, the associated message that is about to be sent is blocked. In the example above, a data store is first created in V_{dfd} . When V_{cfd} receives `insert_data_store_event` from V_{dfd} , it will block the event of sending a message back to V_{dfd} .

Example 2: This rule deletes an `in_out` relationship object from V_{cfd} . An `in_out` relationship object involves an input control flow, an output control flow and a process or control specification object. As a consequence of its deletion, there are two `sends` relationship objects that have to be deleted in the V_{dfd} view. This is illustrated in Figure 5.2. The retrieval action statements specify which `sends` objects are to be deleted along with the `in_out` relationship object. Operations triggered by the `delete_sends_event` are given in Appendix B.4.

```
delete_in_out_event(in_out_object) →  
  CONTEXT:  $V_{cfd}$   
  ACTION: sends_object_1 := retrieve_relationships(sends,  
          “*.data = in_out_object.input AND  
          *.destination = in_out_object.process_name”),  
          SEND delete_sends_event(sends_object_1)  
          To  $V_{dfd}$ ,  
          sends_object_2 := retrieve_relationships(sends,  
          “*.data = in_out_object.output AND
```

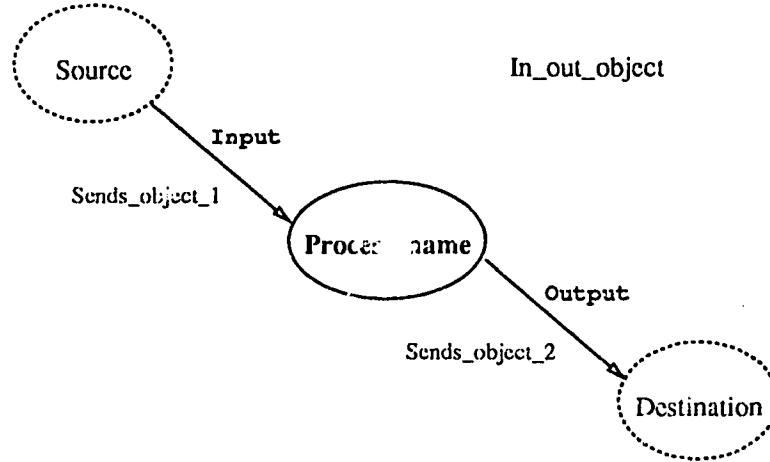


Figure 5.2: Deletion of an in_out Relationship Object

```

        *.source = in_out_object.process_name"),
SEND delete_sends_event(sends_object_2)
    To  $V_{dfd}$ ,
Delete_Relationship(in_out_object, in_out);

```

Example 3: This rule specifies what operations are performed when the value of an attribute is changed in a multi-view context. If a data flow object is a ‘sent’ or ‘stored’ flow, and is changed to a ‘ctrl_signal’, then a corresponding control flow object should be created in V_{cfd} . When the data flow is changed to ‘sent’ or ‘stored’ from ‘ctrl_signal’, the existing control flow object should be deleted.

```

change_flow_type(flow_type) —
CONTEXT:  $V_{dfd}$ 
ACTION: IF self.flow_type = ‘ctrl_signals’ AND flow_type /= ‘ctrl_signals’
        THEN SEND delete_ctrl_flow_event
            To  $V_{cfd}$ ,
        IF self.flow_type /= ‘ctrl_signals’ AND flow_type = ‘ctrl_signals’
        THEN SEND insert_ctrl_flow_event
            To  $V_{cfd}$ ,
        self.flow_type := flow_type;

```


Chapter 6

Integration of Multi-View Environments

In the previous two chapters, model views for a specification environment were developed. Such views capture both multiple representations and operational semantics of object types. This chapter examines the impacts that the multiple view mechanism has on the integration of multiple tools in a specification environments. For this purpose, two traditional integration paradigms are reviewed. One is based on a common repository and the other is based on message-passing. A view-based integration paradigm is then described. In the second part of this chapter, we examine how introducing the multiple views affects the overall Metaview architecture. The enhanced requirements for the Metaview database engine will be specifically addressed.

6.1 Comparison of Integration Paradigms

In Section 2.4, two examples were used to illustrate the basic techniques of tool integration: PCTE's Object Management System and HP Softbench Environment. These two systems represent two major approaches of tool integration: sharing data in a common specification database and sharing functions based on message passing. This section summarizes these two approaches, and compares them with the view-based integration

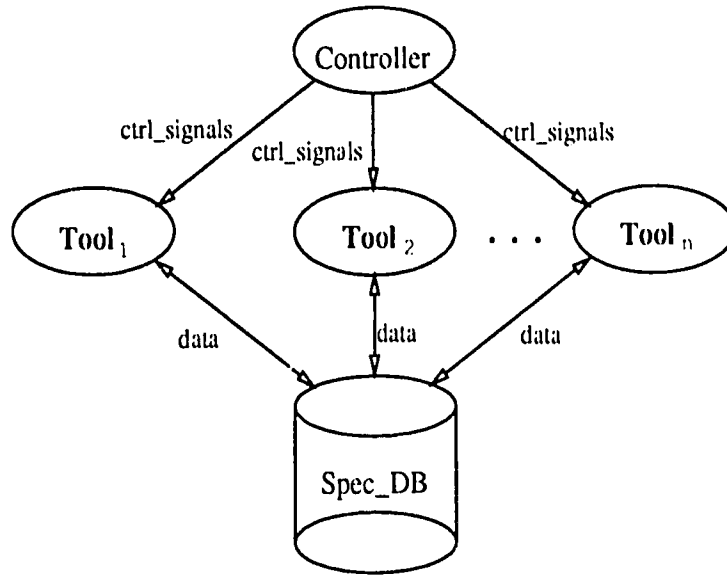


Figure 6.1: Data-based Integration Paradigm

paradigm. The view-based integration was first proposed by D. Garlan[Gar87]. In his case, integrated tools are from a structure-oriented programming environment[GM84, CGG⁺85]. We examine how this paradigm can be adopted for integrating tools in a specification environment.

6.1.1 Review of Two Traditional Integration Paradigms

Integration based on a common database is characterized as a concurrent model, as shown in Figure 6.1.

Tools access a specification database of common structures. A system controller mediates the interaction between the tools and the database. The main advantage is that results of an operation performed in one tool are immediately available to other tools. The specification database provides a means of data communications between tools. However, tools are usually not invoked in an automatic fashion. The system controller is normally a mere coordinating or synchronizing tool. The major disadvantages of this

approach are:

- **Difficulty on environment evolution:** It is hard to modify a tool, because the choice of representations for one tool is restricted by the representations needed by all other tools that access a common specification database. It may be more difficult to add new tools because the existing data representations may be inappropriate. Modifying these data representations can have severe side-effects on the existing tools.
- **Lack of abstraction:** All tools have access to all data representations in the database. There are no abstract interfaces to encapsulate the details of the data used by a given tool.
- **Complexity of data management:** Since a common database has to accommodate various tools which may access the database simultaneously. Managing such a common database can be a very complex task.

The second approach is based on message passing or broadcasting. As shown in Figure 6.2, each tool has its own database or private data representations and works independently. Integration is accomplished by the message communications among the tools. Since each tool chooses its own way of representing data that is appropriate for its particular needs, it is relatively easy to create a new tool and incorporate it with the existing ones. Modifications of the existing tools are also easier because of the tool encapsulation. However, there are also some drawbacks with this paradigm:

- **Lack of data sharing:** Each tool maintains a private representation of the data. It is almost impossible for tools to share the data of different representations. As the result, data redundancy is inevitable.
- **Transformation expenses:** Data transformation is another significant source of expense with this form of integration. For example, a tool may have to copy the outputs of another tool, parse them and convert them to its own representation before it can actually use them.
- **Message restrictions:** Communication requires predefined message formats and well-defined interfaces so that a tool can make use of functional resources of other

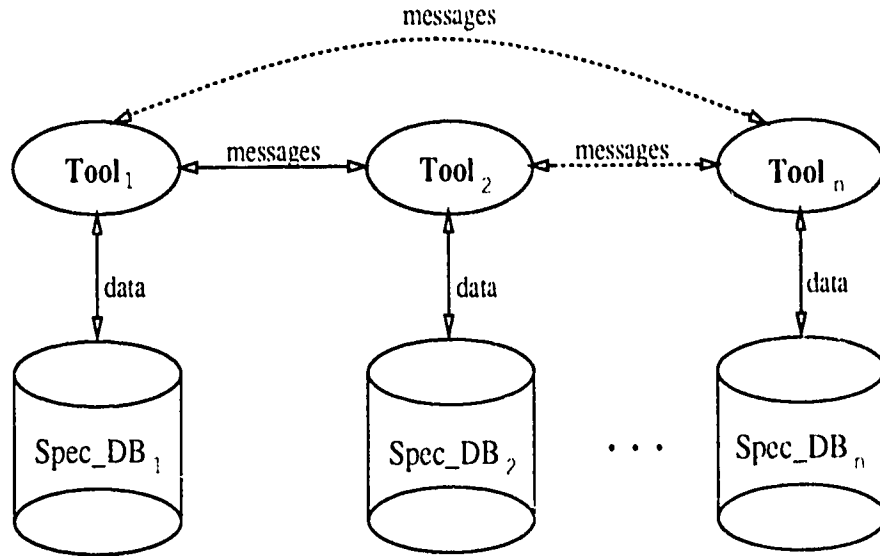


Figure 6.2: Message-based Integration Paradigm

tools. This may incur extra efforts in implementing such a message-based system.

- Unfriendly user interface: Each tool may have its own way of interacting with a user. For example, each may use a different way of invoking its set of commands.

The two paradigms reviewed in this subsection only outlined two typical cases. Many variations to them are possible. For example, a global message handler may be introduced explicitly to the message-based paradigm. Communications among tools may be restricted to one-to-one and uni-directional, i.e. a tool passes data only to its successor when it finishes its processing. In this way, tools are basically chained together sequentially as exemplified by the UNIX pipe facility.

6.1.2 A View-Based Integration Paradigm

The view-based integration is aimed at realizing the advantages of the two traditional approaches while avoiding their disadvantages. In this paradigm, tools operate on a

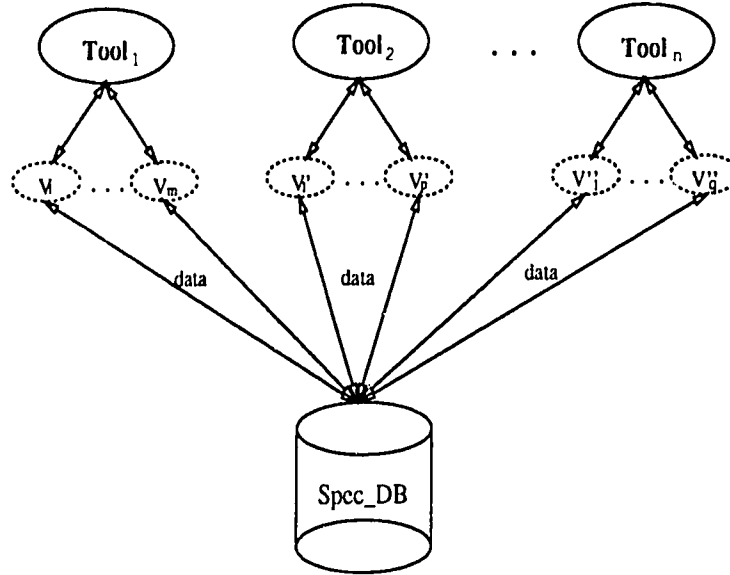


Figure 6.3: View-Based Data Integration

shared database through a collection of views. Each view either defines a virtual representation of the objects contained in the database, or focuses on the different facets of the objects resulted from view merging. Figure 6.3 illustrates that tools are integrated in terms of data through sharing the common specification database. Since each data object in the specification database is accessed through certain views, tools are allowed to use the representations that are tailored to their particular needs. The specification database is basically the synthesis of those views.

On the other hand, tools are also integrated in terms of control, because the operation mappings defined on the views reflect the need for control transfers and consistency maintenance between views. This is illustrated in Figure 6.4.

The advantages of the view-based integration paradigm are:

- Multiple representations: the same object types may be defined in different ways across different views of a specification environment. The different representations

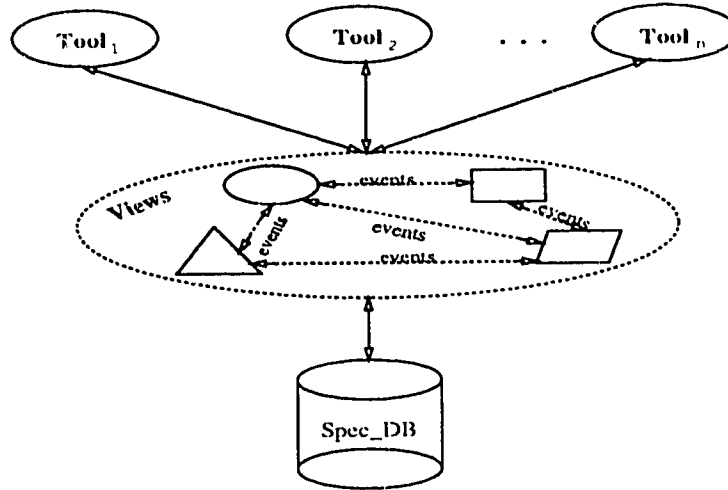


Figure 6.4: View-Based Control Integration

are treated as separate facets of the object types. Since explicit operation mappings are defined at the view definition time to capture the semantics of the object types, the problems of update ambiguity that are common in relational database systems[Kel85] are avoided.

- **Abstraction and encapsulation:** Views provide abstract interfaces. Related environment concepts are usually grouped together to form a view, which can be merged with other views. An environment definer does not have to know the details of the environment definitions of the existing views when creating new ones.
- **Evolution support:** Adding a new tool to the existing system is largely a matter of defining a set of new views. New environment may take advantage of the object types defined in the existing views through view merging and operation mappings.

However, the encapsulation capability provided by the view mechanism is limited.

An environment definer still needs to know how object types are defined in existing views in order to define the consistency-preserving operation mappings. Another major disadvantage of the view-based integration paradigm is that the use of views may degrade the system efficiency. This is because views have imposed an additional level of indirection for operations on database objects. Operations with respect to views have to be translated to ones on database objects. There are also overheads associated with the maintenance of multiple views.

Finally, it should be noted that this section has mainly concentrated on the abstract discussion of the view-based approach of integrating specification environments, implementation-dependent details remain largely unexplored.

6.2 The View-based Metaview Architecture

In accommodating multiple views, the Metaview architecture has to be augmented in several ways. Figure 6.5 highlights the proposed enhancement to the Metaview architecture. Three major components are proposed in support for the view-based environment integration. They are the environment editor, the rule library and the view translator. In the discussion below, we adopt the terminologies given in [Fin93c].

At the meta level, the *Metaview Software Library* has to provide facilities needed to accommodate the requirements of multiple views. At this level, there currently is no view mechanism and the Metaview system simply provides a set of specification object types that can be used to describe a “single-view” software specification environment. The view mechanism we have proposed allows these object types to take different views such that the same specification environment may be described from different perspectives, and this in turn enables a software system being developed to be specified using different software engineering methods. In the previous two chapters, we suggested some extensions to the EDL/ECL languages for defining multiple views of a specification environment. Such extensions complicate the design of the EDL/ECL language compiler. For example, when doing type checking, the EDL/ECL compiler now has to deal with

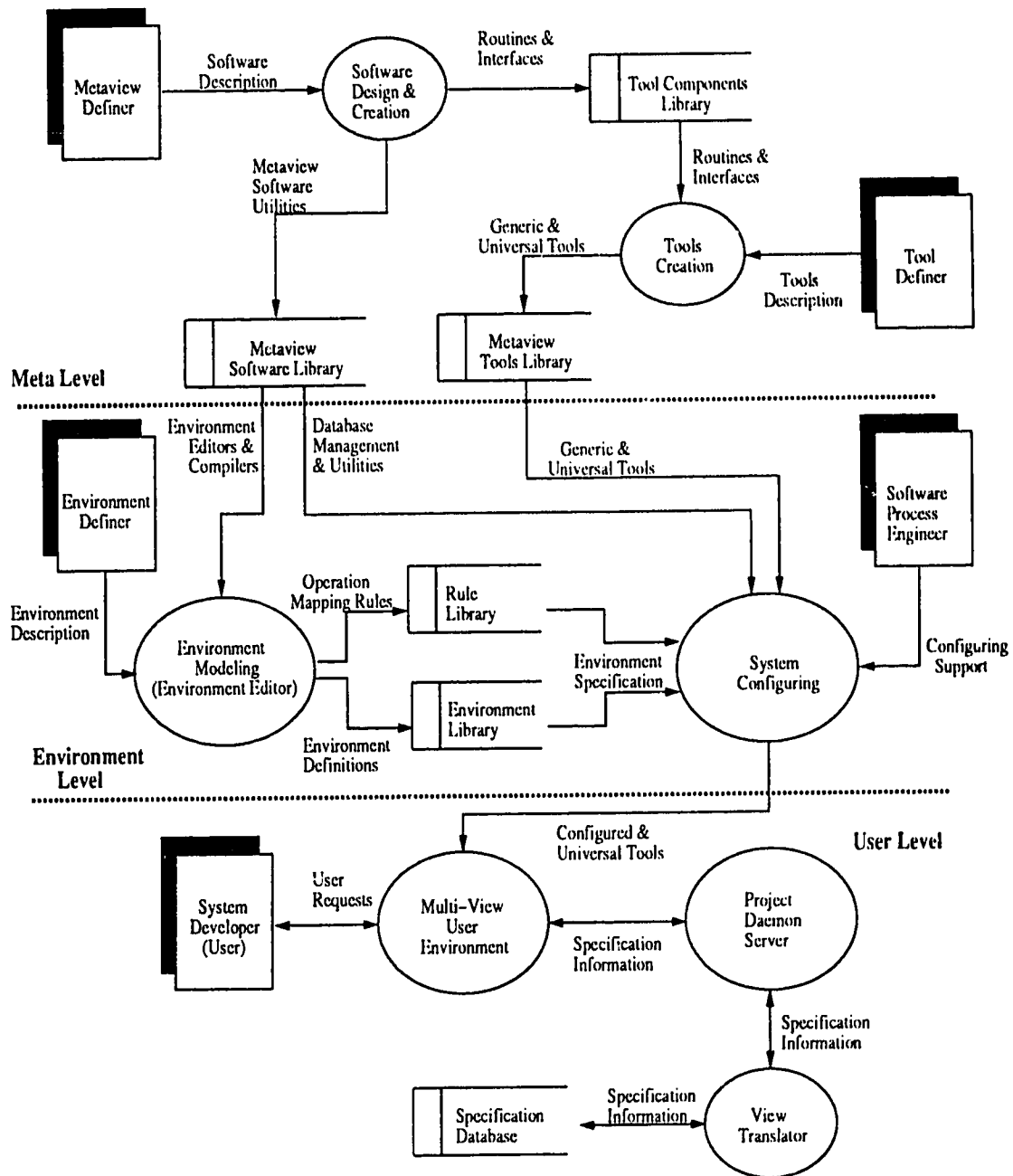


Figure 6.5: Multi-view Architecture of the Metaview System

the scoping problem that was not present in the original EDL/ECL implementations.

At the environment level, the Metaview system now has to deal with environments defined in terms of multiple views. In order to assist an environment definer in defining multi-view specification environments, the Metaview system should provide powerful browsing capabilities that support the visualization, retrieval and editing of the hierarchical views. These capabilities should be present in the *environment editor*, so that the most up-to-date declarative and functional descriptions of the existing views of specification environments can be managed effectively and made available to environment definer in an intelligent way.

In order for different views to work cooperatively, we have introduced the mechanism of operation mappings based on EXA rules. There may also be some other rules that are applicable to a specification environment, such as rules that define software transformations and rules that specify managerial decisions for software project management, etc. To enable the collaborative management of these rules, there is a need to separate the rule definitions from the view definitions. In other words, a *rule library* is needed to store information necessary to guide the control integration of a view-based environment. Information stored in the rule library captures conceptually the dynamic aspects of the augmented Metaview architecture.

The environment library which stores the environment tables compiled from the EDL/ECL definitions must also store the view definitions in assisting the translation of view-based operations to operations on primitive data objects. This translation process is performed by a new function module called the *view translator*. The view translator is proposed as enhancement to the Metaview database engine. The next subsection will examine its functionalities in more detail.

6.2.1 The View Translator

When implementing the view-based integration framework, the most important consideration is how to manage various views including the EXA rules that are associated with

these views. From a data integration perspective, tools access the shared specification database through a set of model views. Information retrieval and update operations that are specified with regard to relevant views must be translated into operations on database objects or the attributes of database objects. From a control integration perspective, operations defined under one view are mapped to those defined under different views. Event-triggering or message passing must be synchronized so that the specification database is maintained to be in a consistent state. The purpose of the view translator is to perform such a translation task and to coordinate the invocation of operations as the results of consistency-preserving operation mappings. The detailed design and implementation of such a view translator is beyond the scope of this thesis. This subsection will discuss briefly how it should be included as part of the Metaview database engine.

In order to describe how such a view translator may facilitate the view translation and event synchronization, it is necessary to review how a system analyst/developer interacts with a specification database. Figure 6.6 shows a part of the Metaview system architecture.

A specification environment is configured from environment tables produced by the EDL/ECL compiler of the Metaview system. A system developer's requests to retrieve or update a specification database are formatted according to the prescribed syntax rules and sent to the *project daemon server*[Fin93b]. The project daemon server isolates the database engine from tools of a specification environment. It is language-independent and works in a server mode. Results from the database operations are reported to the client tool where the requests were originated. The database engine is configured by transforming generic database engine routines into "dedicated" programs that are specific to the defined specification environment. The configured database engine is able to provide commands that manipulate the object types defined in the specification environment and check the consistency of a specification database based on the constraints associated with the object types. Since all the object types are currently represented in a single view, database engine routines always perform operations on specification database objects or their attributes directly.

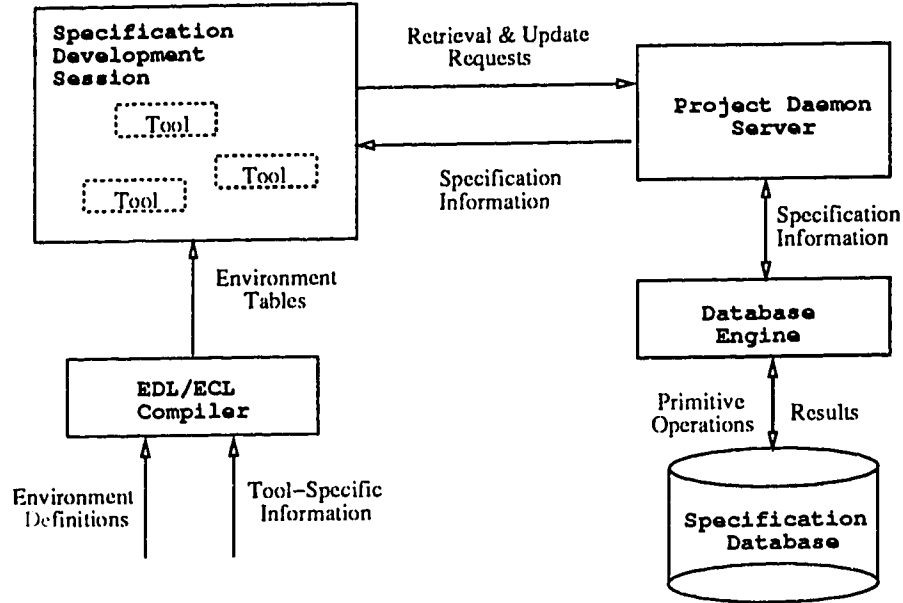


Figure 6.6: Software Specification Development

With the view mechanism added to the system, operations are specified with respect to the context of views. When a request to access a specification database is sent to the project daemon server, the view name is also passed. Similarly, after the operations are carried out by the database engine routines, the results are sent back to the relevant views. The database engine should be augmented by including the view translator as follows:

First, the view translator maintains the two lists, TYPE and ATTR, generated from the view merging process. When the project daemon server receives database access requests, it invokes corresponding database engine routines on the correct view copies of data objects based on the information provided by these two lists. Secondly, the view translator maintains dynamically the EXA rules defined for the views of a specification environment. When consistency-preserving operation mappings are needed for a view operation, appropriate database engine routines are invoked such that automatic control transfer is achieved.

Chapter 7

Conclusions and Future Work

This chapter summarizes the thesis work and highlights the contributions. Some topics for future research are also suggested.

7.1 Thesis Summary

In this thesis, we have investigated multiple view support for an integrated specification environment in the context of Metaview system.

Meyers and Reiss's empirical study[MR92] indicates that multiple views increase a programmer's performance. Although their studies were focused on programming environments such as FIELD[Rei90], the conclusion should be equally true for requirement specification environments. Software requirement information is often developed from different perspectives in a variety of requirement and design specification methods. This makes multiple views especially important at the earlier stages of the software life cycle.

The thesis has focused on specification environments that are generated from formal environment definitions in Metaview. In other words, the scope of the thesis is restricted to issues related to integrating different viewpoints of a software engineering methodology modeled with the EARA metamodel. Since multi-view support is aimed at

providing data sharing and automatic control integration in a specification environment, it is most applicable to the domain of tool integration in the following sense:

- The integration is horizontal, that is, the tools are dedicated to the same phase of a software development life cycle.
- Issues of data and control integrations are addressed. Little emphasis is given to presentation and process integrations.
- “Foreign” (i.e., non-Metaview) tools are not considered for integration.

The motivation for supporting multiple views in a specification environment and the objectives of the thesis is identified in Chapter 1. The main advantage of multiple views is to allow the sharing of information generated from different system perspectives. Different tools that reflect diverse perspectives are configured according to environment definitions represented in different views. Ideally, when changes are made to specifications using one of the tools, operation mapping mechanisms are invoked automatically to keep the specification information consistent for all views. Information redundancy is eliminated as a result of centralized information management. To achieve these objectives within a reasonable scope and time frame, the thesis focused on how to use the EARA metamodel to define multiple representations of software objects and to model the dynamic behaviors of these multi-faceted software objects across a horizontally integrated environment.

In Chapter 2, an example, DFD/CFD environment was introduced to illustrate the difficulties in supporting multiple views. The NIST/ECMA reference model of tool integration was then introduced, and the thesis work was restricted to a subset of the spectrum of problems related to tool integration. Previous work on data integration and control integration was reviewed. Previously, the main approach for data integration was through the use of a shared information repository and control integration was most often accomplished by using message passing or broadcasting facilities. Some efforts in combining data and control integration based on the object-oriented paradigm were

also briefly reviewed. With a view-based approach, these two aspects of tool integration could both be addressed.

Chapter 3 provided a brief overview of the Metaview metasytem. This established a metasytem foundation for multiple views throughout the thesis. Central to the Metaview architecture was the EARA metamodel. The Environment Definition and Constraint Languages were used to express the elements of the EARA metamodel and constraints on their inter-relationships. Multiple views for a specification environment were defined using a direct extension of these two languages.

Chapter 4 concentrated on the static aspects of a view definition. Two levels of views were identified: primitive views and composite views. Different representations of object types were captured in separate primitive views, which were merged to form a composite view. A primitive view was defined as a subschema of a specification schema, which contained a set of entity types, relationship types and aggregate types, along with their associated attributes and constraints. A composite view provided the basis for shared representations of object types in different primitive views and operational semantics of such shared representations. Central to manipulating objects of multiple representations in a common specification database was the idea of view merging. The merging took place between object types of the same name or their synonyms to coalesce the attributes of object types, to resolve definition conflicts, and to establish the associations between object types and their primitive views. Detailed algorithms for this merging process were provided in this chapter. The view approach developed in this chapter was compared with database relational views and object-oriented views.

In Chapter 5, the operation specification in a composite view was discussed. Since the main purpose of view operations was to assist in automatic consistency maintenance of a the specification database when the specification was manipulated with respect to views, the concepts of database state and consistency-preserving operation mappings were defined. Based on these two concepts, syntax for event-triggered, context-based action (EXA) rules was proposed for specifying operation mappings between different primitive views of a composite view. As an example, the EXA rule specification for the

DFD/CFD environment was provided in this chapter.

Chapter 6 examined how to use the view mechanism developed in the previous two chapters to achieve horizontal tool integration in a specification environment. After reviewing more carefully two traditional integration paradigms based on a common repository and message passing, a view-based integration paradigm was proposed. A brief discussion of how the Metaview architecture can be modified to support multiple views was also provided.

7.2 Contributions Of The Thesis

The major contributions of the thesis are as follows:

- A detailed examination of “multiple views” in the context of a specification environment was undertaken. To our knowledge, the concept of multiple views has been explored previously in the context of programming environments. In this thesis, multiple views of a specification environment reflected multiple representations of object types that were used to describe the specification requirements of a software system, while at the same time, such specification information was kept consistent as changes were made to one of these representations.
- The Environment Definition and Constraint Languages (EDL/ECL) were extended to allow both primitive and composite views to be specified. These language extensions permit the expression of multiple views of a specification environment in a formal model. Similar work appeared in Garlan’s PhD thesis[Gar87], in which views in a structured programming environment were captured through formal descriptions of entity types. However, our views also capture other information, such as relationships between entity types, constraints and dynamic behaviors of object types.
- The thesis proposed a new, horizontal view integration mechanism for multi-view support in a specification environment. This contribution has the following two

aspects: statically, an algorithm was proposed which allows lower level primitive views to be merged to form a composite view (i.e., shared representation for each object type defined in a specification environment). Dynamically, a consistency preserving operation mapping mechanism was proposed based on EXA rules, which made it possible for automatic control transfers between views.

- The view mechanism developed in this thesis provides a basis for building a horizontally integrated specification environment using Metaview. The integration paradigm proposed in Chapter 6 dealt with both data integration and control integration. This paradigm is consistent with the NIST/ECMA reference model in the following sense: data integration is achieved sharing a common specification database through a collection of views and control integration is realized through operation mappings that transfer controls between views when messages are passed as triggering events. Such a view-based integration paradigm allows tools in a specification environment to use data representations that are tailored to their particular needs. Based on this integration paradigm, architectural enhancements to the Metaview system were developed. New requirements were proposed for the database engine to translate view operations and to invoke EXA rules associated these views.

7.3 Future Work

While we have, in the thesis, established the ground work for supporting multiple views in a metasystem like Metaview, more detailed design work is needed to elaborate on the proposed architectural design. There are three important issues directly related to extending the view mechanisms:

- *Constraint Integration:* At present when primitive views are merged, constraints are simply merged by forming the union of the set of constraints. Two problems may arise: constraints defined in one view might be conflicting with constraints defined in another and the redundancy may exist among two or more constraints,

which may degrade the systems efficiency significantly. A possible solution for detecting and resolving constraint conflicts is to look into theorem proving techniques. Redundancy problem may not be as simple as recognizing equivalent constraints. Constraints may be overlapping and this is a very difficult condition to detect.

- *Updating Operational Semantics:* As a specification environment evolves, changes to the definitions of object types may cause changes to their associated operational semantics defined using EXA rules. With the current view mechanism, re-writing of these rules is required. Some intelligent support should be present to handle the update needs automatically.
- *Database Management:* With the introduction of multiple views, a specification database stores multiple view copies for each data object. This complicates database management in two ways: First, in order for operations to be performed correctly, either distinct names should be given to different view copies or each view copy should be associated with a view identifier. Secondly, the database should be managed actively in a sense that operation mappings are invoked automatically. How to design and implement such an invocation mechanism remains to be explored.

Other topics that may be interesting for future research include:

- *Presentation Integration:* The thesis has focused on the model views of a specification environment. Presentation integration is not addressed. In the Metaview system, a graphical extension to the EARA metamodel was developed[Sch90]. It supports the definition of graphical representations for software objects defined using the EARA metamodel. The correspondence between graphical objects and multi-view database objects remains largely unexplored.
- *Process Integration:* The effects of multiple view support on software process aspects must be investigated. There are at least two issues that should be explored in this regard. First, the software process modeling capabilities of Metaview must

be examined more extensively. The Metaview approach was used to analyze and describe an example process model called DesignNet[BS^T92]. How to incorporate multiple view support into a process model for integrated project management is still unknown. Secondly, integration of vertical tools needs to be addressed. Vertical integration involves transformations of software products across development life cycle. The Metaview approach was also adopted to define formally the transformations from data flow diagrams to structured charts[BS^T91]. The introduction of multiple views complicates such a transformation process because both source environment and target environment may be generated from multi-view definitions and at least one of the benefits provided by the transformational approach, traceability, is more difficult to realize.

- *Foreign Tools:* In this thesis, we only considered tools that are built within the Metaview system. Tools in Metaview are configured from formal environment definitions and generic tool routines. In order to integrate tools that are developed independently of the Metaview system, there should be a way of describing what and how the specification information is captured. Based on this knowledge, an interface protocol should be defined for communicating with these foreign tools.

Bibliography

- [ANS88] ANSI X3 Technical Committee X3H4. Information Resource Dictionary System, X3.138, 1988.
- [BFW91] Alan W. Brown, Peter H. Feiler, and Kurt C. Wallnau. Past and Future Models of CASE Integration. Technical report, Software Engineering Institute, Carnegie Mellon University, November 1991.
- [BLN86] C. Batini, M. Lenzerini, and S.B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Survey*, pages 323—364, 1986.
- [BM92] Alan W. Brown and John A. McDermid. Learning From IPSE's Mistakes. *IEEE Software*, pages 23–28, March 1992.
- [BMT88] Gerald Boudier, Regis Minot, and Ian Thomas. An Overview of PCTE and PCTE+. *SIGSOFT, Software Engineering Notes*, 13(5):248–257, November 1988.
- [BST91] Germinal Boloix, Paul Sorenson, and Paul Tremblay. On Transformations Using A Metasystem Approach To Software Development. Technical Report TR 91-19, University of Alberta, November 1991.
- [BST92] Germinal Boloix, Paul Sorenson, and Paul Tremblay. Process Modeling Using A Metasystem Approach To Software Specification. Technical Report TR 92-11, University of Alberta, September 1992.
- [Cag90] M.R. Cagen. The IIP SoftBench Environment: An Architecture For a New Generation of Software Tools. *Hewlett-Packard Journal*, 41(3):36–47, June 1990.
- [CAS91] CASE Integration Services Committee (CIS). CIS Base Document V1.0, ATIS Specification, October 1991. also see ANSI X3H4 Working Draft: Information Resource Dictionary System ATIS.

- [CGG⁺85] R. Chandhok, D. Garlan, D. Goldenson, M. Tucker, and L. Miller. Programming Environment Based on Structured Editing: The GNOME Approach. In Anthony S. Wojcik, editor, *Conference Proceedings of the 1985 National Computer Conference, AFIPS*, pages 359–370, Chicago, IL, July 1985.
- [Che76] P.P. Chen. The Entity-Relationship Model — Towards a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.
- [CN92] Minder Chen and Ronald J. Norman. A Framework for Integrated CASE. *IEEE Software*, pages 18–22, March 1992.
- [CNW89] Minder Chen, Jay F. Nunamaker, Junior, and E. Sue Weber. Computer-Aided Software Engineering: Present Status and Future Directions. *DATA BASE*, pages 7–13, Spring 1989.
- [COR91] CORBAS. The Common Request Broker Architecture and Specification. Technical Report OMG Document Number 91.12.1, Revision 1.1, Object Management Group and X/Open, December 1991.
- [D⁺88] U. Dayal et al. The HiPAC Project: Combining Active Databases and Timing Constraints. *SIGMOD RECORD*, 17(1), March 1988.
- [DD92] Dept of Computer Science and Information Systems, Univ. of Jyväskylä and Dept. of Information Processing Science, Univ. of Oulu. MetaPHOR – Metamodeling: Principles, Hypertexts, Objects and Repositories. Project Overview, 1992.
- [DeM79] T. DeMarco. *Structured Systems Analysis*. Prentice Hall, 1979.
- [Dow87] M. Dowson. An Integrated Project Support Environments. In *Proc. Second ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, January 1987.
- [DST86] J.M. Dedourek, P.G. Sorenson, and J.P. Tremblay. Survey of Meta Systems for Information Processing System Specification Environments. *INFOR*, 27(3):311–337, August 1986.
- [Ele90] Electronic Industry Association (EIA). CDIF: Framework for Modeling and Extensibility, EIA-PN2387, 1990.
- [Eur90] European Computer Manufacturers Association. ECMA PC²TE Standards, ECMA-149. ftp'd from stars.rosslyn.unisys.com, December 1990.
- [Fin92] Piotr Findeisen. EARA/GI Model for Metaview, A Reference. Department of Computing Science, University of Alberta, March 1992.
- [Fin93a] Piotr Findeisen. MGED User's Manual. Department of Computing Science, University of Alberta, March 1993.

- [Fin93b] Piotr Findeisen. Project Daemon Reference manual. Department of Computing Science, University of Alberta, March 1993.
- [Fin93c] Piotr Findeisen. The Metaview Software. Department of Computing Science, University of Alberta, March 1993.
- [Fro90] Brian D. Fromme. HP Encapsulator: Bridging the Generation Gap. *Hewlett-Packard Journal*, 41(3):59–68, June 1990.
- [Gar87] David Garlan. *Views for Tools in Integrated Environments*. PhD thesis, Carnegie Mellon University, 1987.
- [GM84] David Garlan and Phillip L. Miller. Gnome: An introductory programming environment based on a family of structure editors. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984.
- [GS79] Chris Gane and Trish Sarson. *Structured Systems Analysis: Tools and Techniques*. Prentice Hall, 1979.
- [Hei90] Sandra Heiler. Object views: Extending the vision. In *Proceedings of Sixth International Conference on Data Engineering*, Los Angeles, CA, February 1990. IEEE Computer Society Press.
- [HKO92] William Harrison, Mansour Kavianpour, and Harold Ossher. Integrating Coarse-Grained and Fine-Grained Tool Integration. Research Report RC 17542 (#77482), IBM, January 1992.
- [HP87] D.J. Hatley and I.A. Pirbhai. *Strategy for Real-Time System Specification*. Dorset House, 1987.
- [HZ88] S. Heiler and S. Zdonik. FUGUE: A Model for Engineering Information Systems and Other Baroque Applications. *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, pages 195–209, June 1988.
- [Kel85] Arthur M. Keller. *Updating Relational Databases Through Views*. PhD thesis, Department of Computer Science, Stanford University, February 1985.
- [Lee92] Jesse K.L. Lee. Implementing ADISSA Transformations in the Metaview Metasystem. Master's thesis, Dept. of Computing Science, University of Alberta, October 1992.
- [MB90] Vaughan Merlyn and Gregory Boone. The Ins and Outs of AD/Cycle. *Data-mation*, pages 59–64, March 1990.
- [McA88] Andrew J. McAllister. *Modeling Concepts for Specification Environments*. PhD thesis, University of Saskatchewan, 1988.

- [MR92] Scott Meyers and Steven P. Reiss. An Empirical Study of Multiple-View Software Development. In *Proc. of the Fifth ACM SIGSOFT Symp. on Software Development Environments*, volume 17, December 1992.
- [MSTD87] A.J. McAllister, P.G. Sorenson, J.P. Tremblay, and J.M. DeDoutre. Constraints for Automatically Generated Requirements Specification Environments. In *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, volume 2, pages 343-352, 1987.
- [Nat91] National Institute of Standards and Technology. Reference Model for Frameworks of Software Engineering Environments. Gaithersburg, Md., 1991. Draft Version 1.5.
- [OV91] M. Tamer Ozsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.
- [Rei85] Steven P. Reiss. PECAN: Program Development Systems That Support Multiple Views. *IEEE Transactions on Software Engineering*, SE-11(3):276-285, March 1985.
- [Rei90] Steve P. Reiss. Connecting Tools Using Message Passing in the FIELD Environment. *IEEE Software*, 7(4):57-66, July 1990.
- [S⁺91] Adrian Smith et al. VOCOS: Voice Conferencing System. Student project for software engineering course at University of Alberta, December 1991.
- [Sag90] J.M. Sagawa. Repository Manager Technology. *IBM Systems Journal*, 29(2):209-227, 1990.
- [Sch90] D. A. Schaefer. Interactive Graphical Tools for Specification Environments. Master's thesis, Dept. of Computational Science, University of Saskatchewan, June 1990.
- [SLTM91] K. Smolander, K. Lyytinen, V.P. Tahvanainen, and P. Marttiin. MetaEdit: A Flexible Graphical Environment for Methodology Modeling. In R. Anderson, J.A. Bubenko, Jr., and A. Solvberg, editors, *Advanced Information Systems Engineering: Third International Conference CAISE'91*, pages 168-193. Springer-Verlag, May 1991.
- [SM88] Paul Sorenson and McAllister. The Metaview System for Many Specification Environments. *IEEE Software*, 5(2):30-38, March 1988.
- [STM92] 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 1992.
- [Tho89a] Ian Thomas. PCTE Interfaces: Supporting Tools in Software Engineering Environments. *IEEE Software*, 6(6):15-23, November 1989.

- [Tho89b] M.I. Thomas. Tool Integration in the PACT Environment. In *Proceedings of International Conference on Software Engineering*, pages 13–22, Los Alamitos, CA, 1989. Computer Society Press.
- [TN92] Ian Thomas and Brian A. Nejme. Definitions of Tool Integration for Environments. *IEEE Software*, pages 29–35, March 1992.
- [TTBG90] Mohamed Tedjini, Ian Thomas, Guy Benoliel, and Ferdinando Gallo. A Query Service for a Software Engineering Database System. In *Proceedings of 4th ACM SIGSOFT Symposium on Software Development Environments*, 1990.
- [UDM88] A.P. Buchmann U. Dayal and D.R. McCarthy. Rules Are Objects Too: A Knowledge Model For An Active, Object-Oriented Database System. In K.R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, pages 127–143. Springer-Verlag, 1988.
- [VHW91] T.F. Verhoef, A.H.M. ter Hofstede, and G.M. Wijers. Structuring Modeling Knowledge for CASE Shells. In R. Anderson, J.A. Bubenko, junior, and A. Solvberg, editors, *Advanced Information Systems Engineering: Third International Conference CAiSE'91*, pages 502–524. Springer-Verlag, May 1991.
- [VJT92] Iris Vessey, Sirkka L. Jarvenpaa, and Noam Tractinsky. Evaluations of CASE Vendor Tools as Methodology Companions. *Communications of The ACM*, 35(4):90–105, April 1992.
- [Was90] A.I. Wasserman. Tool Integration in Software Engineering Environments. In Fred Long, editor, *Software Engineering Environments: Proceedings of International Workshop on Environments*, pages 137–149. Springer-Verlag, 1990.
- [Wil89] Darryl Willick. Database Engine System's Manual. Metaview Documentation, June 1989.
- [WM85] Paul T. Ward and Stephen J. Mellor. *Structured Development for Real-Time Systems*. Yourdon Press, 1985.
- [YC78] E.N. Yourdon and L.L. Constantine. *Structured Design*. Yourdon Press, 1978.

Appendix A

Description of a Two-View (DFD/CFD) Structured Analysis Environment

V_{dfd} : The DFD represented in this view contains four basic concepts: process, data store, data flow, and terminator. While they all have common intrinsic attributes such as **name**, **creation_time**, **description**, etc., some additional attributes are described as follows:

- **process**: the process object in this view consists of three parts: identification, function description and physical location. The identification is of the data type **identifier**, it gives the identification reference of a process object. The function description has the data type **text**, and describes the functions carried out by a process. The physical location may be a program or a function in which the function of the process is implemented. This last attribute takes the **identifier** data type.
- **data_store**: is a place where data is stored by processes. In this view, a data store object is defined to have the following attributes: **reference_identifier**, **access_type**, **number_of_copy** and **primary_key**, and **form** that takes a **string** type with the length of minimum 1 up to 30.
- **data_flow**: is an entity that uses a **description** attribute to describe the data being sent. A attribute named **flow_type** is used to distinguish three types of data flows: those sent between processes, or between a process and a terminator; those being stored away in a **data_store** and those sent as control signals. **flow_type** takes one of the three values: "sent", "stored" and "ctrl_signal".

- **terminator**: usually represents a source or destination of inputs to or outputs from a transaction, e.g., customers, users. It also has the **description** attribute.

There are four relationship types associated with these concepts:

- **sends**: is a relationship that specifies the source, destination and the data being sent. The source and destination can be either a process or a terminator, but a terminator is not allowed to receive data when the source is also a terminator. The data being sent is represented with a **data_flow** object.
- **stores**: describes what data are stored in a data store. So, it specifies a relationship between a **data_store** and a list of **stored_data**.
- **changes/reads**: both represent relationships between a process and a list of **stored_data** that the process is accessing.

In order to model the decomposition scheme, two aggregate concepts are needed: **top_level** and **process_explosion**. There are also relationships that describe how a process in a **top_level** diagram is decomposed into a set of objects in a **process_explosion**. These relationships include **derived_from**, **has_expansion**, **has_parent_boundary**, **has_child_boundary**, and **has_subparts**. The details of these modeling concepts and relationships can be found in [McA88]. For the purpose of this thesis, they will be collectively referred to as the DFD decomposition view.

V_{cf_d}: There are five basic concepts modeled in the CFD represented in this view. **process**, **data_store**, and **terminator** differ slightly from those defined in *V_{dfd}*:

- **process**: the three attributes used to define a process object in the first view are now elided. The intrinsic attribute **description** is used to capture all the information needed to describe a process object.
- **data_store**: **form** attribute is still available, but takes the **text** data type instead of the **string** type in the first view. An attribute named **index** is defined for the same purpose as **primary_key** in *V_{dfd}*.
- **terminator**: is defined the same as the **terminator** in the first view.
- **ctrl_flow**: differs from a data flow in that a control flow also distinguishes discrete data from time-continuous data.
- **ctrl_spec**: specifies the processing of control flows in a real-time system. Since it may be expanded into a STD, a **ctrl_spec** object itself is simply a connector with basic intrinsic attributes.

The DFD decomposition scheme still applies here. Relationship types associated with these concepts include: **has_std**, **in_out** and **access**. **has_std** represents the expansion from a **ctrl_spec** to its associated STD; **in_out** specifies control flows that input to and output from a **process** or a **ctrl_spec**; An **access** relationship object is defined between a process object and a set of control flows with an attribute named **access_type**. This attribute may take two values: “**detects**” and “**modifies**”.

In order to model the STD, the following concepts are needed:

- **state**: is an identifier for any observable mode of behavior. Three types of **state** are identified: **initial**, **final**, and **intermediate**.
- **event**: is a control signal that causes a system to change state. There are three types of **event**: **initial**, **empty**, and **intermediate**.
- **action**: indicates process activation that is taken as a consequence of a particular event.

There is a relationship type named **transition** associated with these concepts. It specifies the event/action sequence that causes one state to change to another. A state can be further decomposed into a lower-level STD. For this purpose, an aggregate type **std_agg** and an associated relationship **has_sub_std** are needed.

Appendix B

Definition of an Integrated DFD/CFD Environment

This appendix gives a complete definition of the two-view DFD specification environment that has been used as the example throughout this thesis. Constraints associated with object types defined in these views are not given in complete details. Readers may refer to [McA88] for the complete definition.

B.1 The Base Views for Structured Analysis Environment

P_VIEW generic_view

ENTITY_TYPE universal GENERIC

ATTRIBUTES (description: text);

ENTITY_TYPE data_object GENERIC IS_A universal;

END_VIEW

P_VIEW value_type_view

VALUE_TYPE time_per_unit

--- data type that is used to describe frequency

RECORD

quantity: integer

unit: (second, minute, hour,
day, week, month, year);

END RECORD;

VALUE_TYPE ref_type

--- data type that is used to describe the reference identification of an object.

--- Assumption is made to have a built-in type constructor concat,

--- and data types range and list.

concat(range(A..Z), list([.]integer[.]));

END_VIEW

```

P_VIEW dfd.decomposition.view
  AGGREGATE_TYPE top_level_dfd
    COMPONENTS (ALL But has_child_boundary,
                has_subparts,
                derived_from);
  AGGREGATE_TYPE process_exploded
    COMPONENTS (ALL);
  RELATIONSHIP_TYPE derived_from
    ROLES (derived_agg, source_agg)
    PARTICIPANTS
      (process_exploded, process_exploded);
  RELATIONSHIP_TYPE has_expansion
    ROLES (parent, child)
    PARTICIPANTS
      (process, process_exploded);
  RELATIONSHIP_TYPE has_parent_boundary
    ROLES (parent, boundary: list)
    PARTICIPANTS
      (process, universal);
  RELATIONSHIP_TYPE has_child_boundary
    ROLES (child, boundary: list)
    PARTICIPANTS
      (process_exploded, universal);
  RELATIONSHIP_TYPE has_subparts
    ROLES (superpart, subpart: list)
    PARTICIPANTS
      (data_flow, data_flow)
      (terminator, terminator)
      (data_store, data_store);
END_VIEW

```

B.2 The DFD Primitive View

P_VIEW V_{dfd}

```
ENTITY_TYPE process IS_A universal;
    ATTRIBUTES (identification: ref_type,
                function_description: text,
                physical_location: identifier);

ENTITY_TYPE data_store IS_A universal
    ATTRIBUTES (ref_id: identifier,
                form: string(1..30),
                access_type: (Read_Only, Read_Write),
                number_of_copy: integer,
                primary_key: identifier);

ENTITY_TYPE data_flow IS_A data_object
    ATTRIBUTES (flow_type: (sent, stored, ctrl_signal));

ENTITY_TYPE terminator IS_A universal;

RELATIONSHIP_TYPE sends
    ROLES (source, data, destination)
    PARTICIPANTS
        (process, data_flow, process | terminator)
        (terminator, data_flow, process)
    ATTRIBUTES (frequency: time_per_unit);

RELATIONSHIP_TYPE stores
    ROLES (store_name, data: list)
    PARTICIPANTS
        (data_store, data_flow)

RELATIONSHIP_TYPE access GENERIC;

RELATIONSHIP_TYPE changes IS_A access
    ROLES (p_name, data: list)
    PARTICIPANTS
        (process, data_flow)
```

```

RELATIONSHIP_TYPE reads IS_A access
    ROLES (p_name, data: list)
    PARTICIPANTS
        (process, data_flow)

CONSTRAINT data_must_be_sent Is
    --- Every data_flow entity must participate
    --- in a sends relationship.
    OBJECTS
        df := (data_flow);
    MUST_HAVE
        (sends: df = *.data);
END;

CONSTRAINT data_must_be_stored Is
    --- Every stored_data entity must participate
    --- in a stores relationship.
    OBJECTS
        ds := (data_flow: *.flow_type = 'stored');
    MUST_HAVE
        (stores: ds IN *.data);
END;

:

END_VIEW

```

B.3 The CFD Primitive View

P_VIEW V_{cfd}

```
ENTITY_TYPE process IS_A universal;
ENTITY_TYPE data_store IS_A universal
    ATTRIBUTES (form: text,
                access_type: (Read_Only, Read_Write),
                index: identifier);
ENTITY_TYPE terminator IS_A universal;
ENTITY_TYPE ctrl_flow IS_A data_object;
ENTITY_TYPE ctrl_spec IS_A universal;

ENTITY_TYPE state IS_A universal;
    ATTRIBUTES (state_type: (initial, final, intermediate)),
ENTITY_TYPE event IS_A universal;
    ATTRIBUTES (event_type: (initial, empty, intermediate)),
ENTITY_TYPE action IS_A universal;

AGGREGATE_TYPE std_agg
    COMPONENTS (state, event, action, transition, has_sub_std);

RELATIONSHIP_TYPE access
    ROLES (process_name, data: list)
    PARTICIPANTS
        (process, ctrl_flow)
    ATTRIBUTES (access_type: (detects, modifies));
RELATIONSHIP_TYPE in_out
    ROLES (input: list, process_name, output: list)
    PARTICIPANTS
        (ctrl_flow, ctrl_spec, ctrl_flow)
        (ctrl_flow, process, ctrl_flow);
RELATIONSHIP_TYPE has_std
    ROLES (cspec_name, std_name)
    PARTICIPANTS
        (ctrl_spec, std_agg)
RELATIONSHIP_TYPE transition
    ROLES (from_state_name, cflow_name, action_name, to_state_name)
    PARTICIPANTS
```



```

                (state, event, action, state)
RELATIONSHIP_TYPE has_sub_std
    ROLES (state_name, std_name)
    PARTICIPANTS
        (state, std_agg)

CONSTRAINT process_must_participate Is
    --- Every process entity must participate
    --- in in_out relationship.
    OBJECTS
        p := (process);
    MUST_HAVE
        (in_out: p = *.process_name);
END;

:

END_VIEW

```

B.4 The Composite View for the DFD/CFD Environment

C_VIEW dfd/cfd.view

DEFINITION:

BASE_VIEW: dfd_decomposition_view,
generic_view,
value_type_view;

MERGE V_{dfd} , V_{cfd} ;

SYNONYM (ctrl_flow, data_flow)
(ctrl_spec, process)
(action, process)
(data_store.index, data_store.primary_key);

OPERATION:

insert_data_store_event(form_value) →
CONTEXT: V_{dfd}
ACTION: insert_entity(self, data_store),
self.form := form_value,
SEND insert_data_store_event(string_to_text(form_value))
TO V_{cfd} ;

insert_data_store_event(form_value) →
CONTEXT: V_{cfd}
ACTION: insert_entity(self, data_store),
self.form := form_value,
SEND insert_data_store_event(text_to_string(form_value))
TO V_{dfd} ;

insert_data_flow_event(flow_type) →
CONTEXT: V_{dfd}
ACTION: IF (flow_type = 'ctrl_signal') THEN
SEND insert_ctrl_flow_event TO V_{cfd} ,
Insert_Entity(self, data_flow),
self.flow_type := flow_type,
CHECK(data_must_be_sent);

insert_ctrl_flow_event →
CONTEXT: V_{cfd}
ACTION: Insert_Entity(self, ctrl_flow),

```

        SEND insert_data_flow_event('ctrl.signal') To  $V_{dfd}$ ,
        CHECK(data_must_flow);
insert_changes_event(process_name, data_name) →
    CONTEXT:  $V_{dfd}$ 
    ACTION: Insert_Relationship(self, changes),
            self.p_name := process_name,
            self.data := data_name,
            SEND insert_access_event('modifies')
              To  $V_{cfd}$ ;
insert_reads_event(process_name, data_name) →
    CONTEXT:  $V_{dfd}$ 
    ACTION: Insert_Relationship(self, reads),
            self.p_name := process_name,
            self.data := data_name,
            SEND insert_access_event('detects')
              To  $V_{cfd}$ ;
insert_access_event(access_type) →
    CONTEXT:  $V_{cfd}$ 
    ACTION: Insert_Relationship(self, access),
            self.access_type := access_type,
            IF (access_type = 'detects') THEN
                SEND insert_reads_event(self.process_name, self.data)
                  To  $V_{dfd}$ 
            ELSE
                SEND insert_changes_event(self.process_name, self.data)
                  To  $V_{dfd}$ ;
insert_sends_event(data_sent, from, to, freq_value) →
    CONTEXT:  $V_{dfd}$ 
    ACTION: Insert_Relationship(self, sends),
            self.frequency := freq_value,
            self.data := data_sent,
            self.source := from,
            self.destination := to,
            SEND insert_in_out_event(',', from, data_sent)
              To  $V_{cfd}$ 
            SEND insert_in_out_event(data_sent, to, ',')
              To  $V_{cfd}$ ;
insert_in_out_event(in_flow, process_cspec, out_flow) →
    CONTEXT:  $V_{cfd}$ 

```

```

ACTION:  IF (in_flow = '') THEN
        {
            sends_object_1 := retrieve_relationships( sends,
                                                    "**.destination = process_cspec"),
            Insert_Entity(sends_object_1.data, ctrl_flow),
            Insert_Relationship(self, in_out),
            self.input := sends_object_1.data,
            self.output := out_flow,
            self.process_name := process_cspec,
        }
    IF (out_flow = '') THEN
        {
            sends_object_1 := retrieve_relationships(sends,
                                                    "**.source = process_cspec"),
            Insert_Entity(sends_object_2.data, ctrl_flow),
            Insert_Relationship(self, in_out),
            self.input := in_flow,
            self.output := sends_object_2.data,
            self.process_name := process_cspec,
        }
    ELSE
        {
            Insert_Relationship(self, in_out),
            self.input := in_flow,
            self.output := out_flow,
            self.process_name := process_cspec,
            SEND insert_sends_event(in_flow, '', process_cspec, '')
                To  $V_{dfd}$ 
            SEND insert_sends_event(out_flow, process_cspec, '', '')
                To  $V_{dfd}$ 
        };
insert_transition_event(from, event_name, actions, to) →
CONTEXT:  $V_{cfd}$ 
ACTION:  Insert_Relationship(self, transition),
        self.from_state_name := from,
        self.cflow_name := event_name,
        self.action_name := actions,
        self.to_state_name := to,
        SEND insert_reads_event(actions, event_name)

```

```

                                To  $V_{dfd}$ ,
delete_data_flow_event →
    CONTEXT:  $V_{dfd}$ 
    ACTION: Delete_Entity(self, data_flow),
            IF self.flow_type = 'ctrl_signal' THEN
                SEND delete_ctrl_flow_event To  $V_{cfd}$ ;
delete_ctrl_flow_event →
    CONTEXT:  $V_{cfd}$ 
    ACTION: Delete_Entity(self, ctrl_flow),
            SEND delete_data_flow_event To  $V_{dfd}$ ;
delete_changes_event(changes_object) →
    CONTEXT:  $V_{dfd}$ 
    ACTION: Delete_Relationship(changes_object, changes),
            access_object := retrieve_relationships(access,
                "*.process_name = changes_object.p_name AND
                *.data = changes_object.data");
            SEND delete_access_event(access_object) To  $V_{cfd}$ ;
delete_reads_event(reads_object) →
    CONTEXT:  $V_{dfd}$ 
    ACTION: Delete_Relationship(reads_object, reads),
            access_object := retrieve_relationships(access,
                "*.process_name = reads_object.p_name AND
                *.data = reads_object.data");
            SEND delete_access_event(access_object) To  $V_{cfd}$ ;
delete_access_event(access_object) →
    CONTEXT:  $V_{cfd}$ 
    ACTION: Delete_Relationship(access_object, access),
            IF (access_object.access_type = 'detects') THEN
                reads_object := retrieve_relationships(reads,
                    "*.p_name = access_object.process_name AND
                    *.data = access_object.data");
                SEND delete_reads_event(reads_object) To  $V_{dfd}$ ,
            ELSE
                changes_object := retrieve_relationships(changes,
                    "*.p_name = access_object.process_name AND
                    *.data = access_object.data");
                SEND delete_changes_event(changes_object) To  $V_{dfd}$ ;
delete_sends_event(sends_obj →
    CONTEXT:  $V_{dfd}$ 

```

```

ACTION: Delete_Relationship(sends_obj, sends),
      SEND delete_in_out_event('', sends_obj.source, sends_obj.data)
        To  $V_{cfd}$ 
      SEND delete_in_out_event(sends_obj.data, sends_obj.destination, '')
        To  $V_{cfd}$ ;
delete_in_out_event(in_out_object) →
  CONTEXT:  $V_{cfd}$ 
  ACTION: sends_object_1 := retrieve_relationships(sends,
    ".*data = in_out_object.input AND
    *.destination = in_out_object.process_name"),
  SEND delete_sends_event(sends_object_1)
    To  $V_{dfd}$ ,
  sends_object_2 := retrieve_relationships(sends,
    ".*data = in_out_object.output AND
    *.source = in_out_object.process_name"),
  SEND delete_sends_event(sends_object_2)
    To  $V_{dfd}$ ,
  Delete_Relationship(in_out_object, in_out);
delete_transition_event(transition_object) →
  CONTEXT:  $V_{cfd}$ 
  ACTION: reads_object := retrieve_relationships(reads,
    ".*p_name = transition_object.action_name AND
    *.data = transition_object.cflow_name"),
  SEND delete_reads_event(reads_object) To  $V_{dfd}$ ,
  Delete_Relationship(transition_object, transition);
change_access_type_event(new_access_type) →
  CONTEXT:  $V_{cfd}$ 
  ACTION: self.access_type := new_access_type,
  IF (new_access_type = 'detects') THEN
  {
    changes_obj := retrieve_relationships(changes,
      ".*p_name = self.process_name AND
      *.data = self.data")
    SEND (insert_reads_event(self.process_name, self.data),
      delete_changes_event(changes_obj))
      To  $V_{dfd}$ 
  }
  ELSE
  {

```

```

        reads_obj := retrieve_relationships(reads,
            "*.p_name = self.process_name AND
            *.data = self.data")
        SEND (insert_changes_event(self.process_name, self.data),
            delete_reads_event(reads_obj))
        To  $V_{dfd}$ ;
    }
change_form(form_value) →
    CONTEXT:  $V_{dfd}$ 
    ACTION: self.form := form_value,
        SEND change_form(string_to_text(form_value))
        To  $V_{cfd}$ ;
change_form(form_value) →
    CONTEXT:  $V_{cfd}$ 
    ACTION: self.form := form_value,
        SEND change_form(text_to_string(form_value))
        To  $V_{dfd}$ ;
change_flow_type(flow_type) →
    CONTEXT:  $V_{dfd}$ 
    ACTION: IF (self.flow_type = 'ctrl_signals' AND
        flow_type /= 'ctrl_signals')
        THEN SEND delete_ctrl_flow_event To  $V_{cfd}$ ,
        IF (self.flow_type /= 'ctrl_signals' AND
        flow_type = 'ctrl_signals')
        THEN SEND insert_ctrl_flow_event To  $V_{cfd}$ ,
        self.flow_type := flow_type;
END_VIEW dfd/cfd.view

```