



# Product Information Announcement

☐ New Release    ☒ Revision    ☐ Update    ☐ New Mail Code

---

Title

**MCP/AS InfoExec™ Semantic Information Manager (SIM) Object Definition Language (ODL) Programming Guide (8600 0189-101)**

This announces a retitling and reissue of the *ClearPath HMP NX and A Series InfoExec™ Semantic Information Manager (SIM) Object Definition Language (ODL) Programming Guide*. No new technical changes have been introduced since the HMP 1.0 and SSR 43.2 release in June 1996.

To order a Product Information Library CD-ROM or paper copies of this document

- United States customers, call Unisys Direct at 1-800-448-1424.
- Customers outside the United States, contact your Unisys sales office.
- Unisys personnel, order through the electronic Book Store at <http://www.bookstore.unisys.com>.

Comments about documentation can be sent through e-mail to **[doc@unisys.com](mailto:doc@unisys.com)**.

---

Announcement only:

Announcement and attachments:  
AS123

System: MCP/AS  
Release: HMP 4.0 and SSR 45.1  
Date: June 1998  
Part number: 8600 0189-101



# MCP/AS

## **UNISYS**

# InfoExec™ Semantic Information Manager (SIM) Object Definition Language (ODL)

## Programming Guide

Copyright © 1998 Unisys Corporation.

All rights reserved.

Unisys is a registered trademark of Unisys Corporation.

HMP 4.0 and SSR 45.1

June 1998

Priced Item

Printed in USA  
8600 0189-101

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product or related information described herein is only furnished pursuant and subject to the terms and conditions of a duly executed agreement to purchase or lease equipment or to license software. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, special, or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

RESTRICTED – Use, reproduction, or disclosure is restricted by DFARS 252.227–7013 and 252.211–7015/FAR 52.227–14 & 52.227-19 for commercial computer software.

Correspondence regarding this publication should be forwarded to Unisys Corporation either by using the Business Reply Mail form at the back of this document or by addressing remarks to Software Product Information, Unisys Corporation, 25725 Jeronimo Road, Mission Viejo, CA 92691–2792 U.S.A.

Comments about documentation can also be sent through e-mail to **doc@unisys.com**.

Unisys and ClearPath are registered trademarks of Unisys Corporation.

InfoExec is a trademark of Unisys Corporation.

Personal Workstation<sup>2</sup> is a trademark of Unisys Corporation.

All other terms mentioned in this document that are known to be trademarks or service marks have been appropriately capitalized. Unisys Corporation cannot attest to the accuracy of this information. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

# Contents

|  |            |
|--|------------|
| <b>About This Guide .....</b>                        | <b>xv</b>  |
| <br><b>Section 1. Introduction to ODL</b>            |            |
| <b>Databases .....</b>                               | <b>1-1</b> |
| <b>Database Management Systems .....</b>             | <b>1-1</b> |
| <b>Data Models .....</b>                             | <b>1-2</b> |
| <b>The Semantic Information Manager (SIM) .....</b>  | <b>1-2</b> |
| <b>The InfoExec System .....</b>                     | <b>1-3</b> |
| <b>Entities and Objects .....</b>                    | <b>1-3</b> |
| <b>Database Schemas .....</b>                        | <b>1-3</b> |
| <b>Database Schema Development .....</b>             | <b>1-4</b> |
| Direct Schema Development .....                      | 1-4        |
| SIM Design Assistant (SDA) .....                     | 1-4        |
| Advanced Data Dictionary System (ADDS) .....         | 1-5        |
| <b>ODL Declarations .....</b>                        | <b>1-6</b> |
| Kinds of ODL Declarations .....                      | 1-6        |
| Categories of ODL Constructs .....                   | 1-6        |
| <b>Example Database Schema Development .....</b>     | <b>1-7</b> |
| <br><b>Section 2. SIM Database Design Guidelines</b> |            |
| <b>Initial Considerations .....</b>                  | <b>2-1</b> |
| The Application Life Cycle .....                     | 2-2        |
| Development Methodologies .....                      | 2-4        |
| Program Development .....                            | 2-4        |
| Database Development .....                           | 2-4        |
| SIM Database Development .....                       | 2-5        |
| Prototyping .....                                    | 2-5        |
| <b>SIM Database Design .....</b>                     | <b>2-6</b> |
| Overview .....                                       | 2-6        |
| Logical Design .....                                 | 2-7        |
| Step 1: Identify Things and Their Properties .....   | 2-7        |
| Identifying Classes .....                            | 2-8        |
| Identifying Attributes .....                         | 2-8        |
| Attribute Placement .....                            | 2-11       |
| Class Attributes .....                               | 2-11       |
| Naming Conventions .....                             | 2-11       |
| Attribute Types .....                                | 2-12       |
| Relationships .....                                  | 2-12       |
| Step 2: Generalize and Specialize .....              | 2-14       |

|  |             |
|--|-------------|
| Forming Hierarchies .....                              | 2-14        |
| Critical Examples .....                                | 2-15        |
| Role Relationships .....                               | 2-15        |
| Redundant Data .....                                   | 2-16        |
| Step 3: Add Semantics .....                            | 2-18        |
| Adding Constraints .....                               | 2-19        |
| Attribute Options .....                                | 2-19        |
| User-Defined Types .....                               | 2-20        |
| UNIQUE Indexes .....                                   | 2-21        |
| Verifies .....   | 2-21        |
| Security .....   | 2-23        |
| Step 4: Evaluate and Refine .....                      | 2-24        |
| Matching Requirements to the Schema .....              | 2-24        |
| EVA/Attribute Ratios .....                             | 2-24        |
| Information = Data + Rules .....                       | 2-25        |
| Unnecessary Classes .....                              | 2-26        |
| Nonrequired Attributes .....                           | 2-27        |
| Fringe Conditions .....                                | 2-27        |
| Many-to-Many Relationships .....                       | 2-28        |
| Isolated Islands of Data .....                         | 2-29        |
| Tradeoffs and Special Cases .....                      | 2-29        |
| Interface Design .....                                 | 2-31        |
| Step 1: Identify Ad Hoc Query Requirements .....       | 2-31        |
| Capturing Queries .....                                | 2-32        |
| Update Queries .....                                   | 2-32        |
| Retrieval Queries .....                                | 2-32        |
| Step 2: Identify Programmatic Query Requirements ..... | 2-33        |
| Capturing Queries .....                                | 2-33        |
| Update Queries .....                                   | 2-33        |
| Data Types and Formats .....                           | 2-34        |
| Step 3: Evaluate and Refine .....                      | 2-34        |
| Physical Design .....                                  | 2-35        |
| Step 1: Identify Capacity Requirements .....           | 2-35        |
| Step 2: Identify Performance Requirements .....        | 2-36        |
| Step 3: Identify Operational Requirements .....        | 2-37        |
| Backup and Recovery .....                              | 2-37        |
| Schema Control .....                                   | 2-38        |
| Schema Evolution .....                                 | 2-38        |
| Step 4: Evaluate and Refine .....                      | 2-39        |
| <b>The SIM Data Model beyond Databases .....</b>       | <b>2-40</b> |

## Section 3. Basic ODL Considerations

|   |     |
|---|-----|
| Kinds of ODL Declarations .....                 | 3-1 |
| Creating and Managing a Schema File .....       | 3-3 |
| Syntax Rules .....                              | 3-3 |
| Language Conventions .....                      | 3-4 |
| The Parts of a Declaration .....                | 3-4 |
| Rules for Using SIM Identifiers .....           | 3-5 |
| Kinds of Comments You Can Use in a Schema ..... | 3-6 |

|  |      |
|--|------|
| Official Comments .....  | 3-6  |
| Escape Comments .....  | 3-6  |
| <b>Constant Data Values</b> .....  | 3-7  |
| Integer .....  | 3-7  |
| Real .....   | 3-8  |
| Character .....  | 3-8  |
| Graphic Character Format .....   | 3-8  |
| Hexadecimal Character Format .....                                       | 3-9  |
| Kanji .....  | 3-9  |
| Boolean .....  | 3-9  |
| Time .....   | 3-10 |
| Date .....   | 3-10 |
| Double .....   | 3-11 |
| String .....   | 3-11 |
| Graphic String Format .....  | 3-11 |
| Hexadecimal String Format .....  | 3-12 |
| Considerations for Using Graphic and<br>Hexadecimal String Formats ..... | 3-12 |
| <b>ORGANIZATION Database</b> .....                                       | 3-14 |
| General Description .....  | 3-14 |
| Schema .....   | 3-17 |

## Section 4. Types

|  |      |
|--|------|
| <b>Distinction between Types and Classes</b> ..... | 4-1  |
| <b>Basic Kinds of Data Types</b> .....             | 4-2  |
| <b>Specifying Data Types</b> .....                 | 4-2  |
| Primitive Types .....                              | 4-3  |
| INTEGER .....                                      | 4-5  |
| REAL .....   | 4-5  |
| CHAR .....   | 4-6  |
| KANJI .....  | 4-7  |
| BOOLEAN .....                                      | 4-7  |
| TIME .....   | 4-7  |
| DATE .....   | 4-8  |
| NUMBER .....                                       | 4-8  |
| STRING .....                                       | 4-9  |
| Constructor Types .....                            | 4-12 |
| Compound .....                                     | 4-13 |
| SYMBOLIC .....                                     | 4-15 |
| User-Defined Type Name .....                       | 4-16 |
| <b>User-Defined Types</b> .....                    | 4-18 |
| User-Defined Type Declaration .....                | 4-18 |
| Examples of User-Defined Type Declarations .....   | 4-19 |

## Section 5. Classes

|                                  |     |
|----------------------------------|-----|
| <b>General Concepts</b> .....    | 5-2 |
| Entity Relationships .....       | 5-2 |
| Generalization Hierarchies ..... | 5-2 |

|   |      |
|---|------|
| <b>Defining Classes</b>                     | 5-4  |
| Base Class Declaration                      | 5-4  |
| Subclass Declaration                        | 5-6  |
| Class Attributes                            | 5-8  |
| Attributes                                  | 5-10 |
| Data-Valued Attributes (DVAs)               | 5-10 |
| Entity-Valued Attributes (EVAs)             | 5-16 |
| <br><b>Section 6. Indexes</b>               |      |
| <br>Purposes for Declaring Indexes          | 6-1  |
| Index Declaration                           | 6-2  |
| Restrictions on All Indexes                 | 6-3  |
| Restrictions on UNIQUE Indexes              | 6-4  |
| Guidelines for Using Indexes                | 6-4  |
| <br><b>Section 7. Verifies</b>              |      |
| <br>When to Declare a Verify                | 7-1  |
| Techniques for Using Verifies               | 7-1  |
| Verify Declaration                          | 7-2  |
| How Verifies Are Used                       | 7-4  |
| Locking Implications in Using Verifies      | 7-4  |
| When Verifies Are Valid                     | 7-5  |
| Restriction on Adding or Changing a Verify  | 7-5  |
| Examples of Verify Declarations             | 7-5  |
| <br><b>Section 8. Security</b>              |      |
| <br>Accesses                                | 8-1  |
| Access Declaration                          | 8-2  |
| ALLDB Access                                | 8-5  |
| Class Attribute Access                      | 8-6  |
| Attribute Access                            | 8-7  |
| Examples of Access Declarations             | 8-7  |
| Permissions                                 | 8-9  |
| Permission Declaration                      | 8-10 |
| Examples of Permission Declarations         | 8-12 |
| <br><b>Section 9. DMSII Mapping Options</b> |      |
| <br>Declaring DMSII Mapping Options         | 9-1  |
| Syntax for Declaring Mapping Options        | 9-2  |
| Global Mapping Options                      | 9-3  |
| Restrictions and Ramifications              | 9-5  |
| Code File Titles                            | 9-5  |
| Defaults                                    | 9-5  |
| Options                                     | 9-5  |
| Parameters                                  | 9-6  |



|   |      |
|---|------|
| Audit Trail Attributes .....                        | 9-6  |
| Control Files .....                                 | 9-6  |
| Database Physical Options .....                     | 9-7  |
| Performance Considerations .....                    | 9-7  |
| Guard Files with SIM Databases .....                | 9-9  |
| Example Declaration of Global Mapping Options ..... | 9-12 |
| <b>Base Class Mapping Options</b> .....             | 9-13 |
| SURROGATE Option .....                              | 9-13 |
| TIMESTAMP Scheme .....                              | 9-14 |
| Attribute Scheme .....                              | 9-15 |
| Data Set Options for a Base Class .....             | 9-17 |
| <b>Attribute Mapping Options</b> .....              | 9-20 |
| Class Attribute Mapping .....                       | 9-21 |
| Single-Valued DVA Mapping .....                     | 9-22 |
| Default Single-Valued DVA Mapping .....             | 9-22 |
| Single-Valued DVA Mapping Options .....             | 9-23 |
| Multivalued DVA Mapping .....                       | 9-28 |
| Default Multivalued DVA Mapping .....               | 9-28 |
| Multivalued DVA Mapping Options .....               | 9-28 |
| EVA Mapping .....                                   | 9-38 |
| Default EVA Mapping .....                           | 9-39 |
| Single-Valued EVA Mapping Options .....             | 9-39 |
| Multivalued EVA Mapping Options .....               | 9-40 |
| One-to-One Relationship Mapping .....               | 9-41 |
| Foreign-Key Mapping .....                           | 9-41 |
| Common Data Set Mapping .....                       | 9-42 |
| One-to-Many Relationship Mapping .....              | 9-43 |
| Foreign-Key Mapping .....                           | 9-43 |
| Common Data Set Mapping .....                       | 9-45 |
| Many-to-Many Relationship Mapping .....             | 9-46 |
| <b>Subclass Mapping Options</b> .....               | 9-47 |
| MAPPING Option .....                                | 9-47 |
| VARIABLE-FORMAT Option .....                        | 9-48 |
| DISJOINT-DATASET Option .....                       | 9-50 |
| Data Set Options for a Subclass .....               | 9-51 |
| <b>Index Mapping Options</b> .....                  | 9-53 |
| DUPLICATES Option .....                             | 9-54 |
| SET-OPTIONS Option for an Index .....               | 9-55 |

## Section 10. Dictionary Options

|  |      |
|--|------|
| <b>Using a Dictionary-Options Declaration</b> .....      | 10-2 |
| Advantages .....   | 10-2 |
| Effect on a Database .....                               | 10-3 |
| <b>Example of a Dictionary-Options Declaration</b> ..... | 10-4 |

## Section 11. SIM Utility

|  |       |
|--|-------|
| <b>Database Types</b> .....                          | 11-1  |
| Validated Databases .....                            | 11-2  |
| Generated Databases .....                            | 11-2  |
| Viewed Databases .....                               | 11-3  |
| <b>Executing the SIM Utility</b> .....               | 11-4  |
| Program Parameter .....                              | 11-4  |
| Input Files .....                                    | 11-4  |
| Output Files .....                                   | 11-4  |
| Remote Execution .....                               | 11-4  |
| Batch Execution .....                                | 11-5  |
| Status File .....                                    | 11-6  |
| Conflicting Executions .....                         | 11-6  |
| Program Restarts .....                               | 11-6  |
| <b>Utility Commands</b> .....                        | 11-7  |
| ADD Command .....                                    | 11-7  |
| CHANGE Command .....                                 | 11-9  |
| UPDATE Command .....                                 | 11-13 |
| Updating Software Levels .....                       | 11-15 |
| Copying a Database .....                             | 11-16 |
| REINITIALIZE Command .....                           | 11-16 |
| Using the REINITIALIZE Command .....                 | 11-17 |
| Reinitializing a Database that Contains Verifies ... | 11-17 |
| LIST Command .....                                   | 11-18 |
| LOAD Command .....                                   | 11-19 |
| <b>Utility Files</b> .....                           | 11-20 |
| CARD File .....                                      | 11-20 |
| DASDL File .....                                     | 11-20 |
| DDLRESULTS File .....                                | 11-21 |
| LINE File .....                                      | 11-22 |
| <b>Utility Options</b> .....                         | 11-23 |
| LIMIT Option .....                                   | 11-23 |
| LIST Option .....                                    | 11-23 |
| WARNSUPR Option .....                                | 11-24 |

## Section 12. Changing a SIM Database Schema

|  |       |
|--|-------|
| <b>Understanding Terms</b> .....                   | 12-2  |
| <b>How Schema Changes Affect Queries</b> .....     | 12-3  |
| <b>User-Defined Type Changes</b> .....             | 12-4  |
| <b>Base Class Changes</b> .....                    | 12-5  |
| <b>Subclass Changes</b> .....                      | 12-6  |
| <b>Class Attribute Changes</b> .....               | 12-6  |
| <b>Data-Valued Attribute (DVA) Changes</b> .....   | 12-8  |
| <b>Entity-Valued Attribute (EVA) Changes</b> ..... | 12-11 |
| <b>Index Changes</b> .....                         | 12-12 |
| <b>Verify Changes</b> .....                        | 12-13 |
| <b>Access Changes</b> .....                        | 12-13 |
| <b>Permission Changes</b> .....                    | 12-14 |
| <b>DMSII Mapping Option Changes</b> .....          | 12-14 |

|  |       |
|--|-------|
| Dictionary Option Changes .....                | 12-16 |
| Circumventing Schema Change Restrictions ..... | 12-16 |

## Appendix A. SIM Reserved Words

## Appendix B. Syntax for Utility Commands and ODL

|  |     |
|--|-----|
| SIM Utility Commands .....             | B-1 |
| Object Definition Language (ODL) ..... | B-3 |

## Appendix C. Understanding Railroad Diagrams

|  |     |
|--|-----|
| <b>Railroad Diagram Concepts</b> .....                   | C-1 |
| Paths .....  | C-1 |
| Constants and Variables .....                            | C-2 |
| Constraints .....  | C-3 |
| Vertical Bar .....                                       | C-3 |
| Percent Sign .....                                       | C-3 |
| Right Arrow .....  | C-3 |
| Required Item .....                                      | C-4 |
| User-Selected Item .....                                 | C-4 |
| Loop .....   | C-5 |
| Bridge .....   | C-5 |
| <b>Following the Paths of a Railroad Diagram</b> .....   | C-6 |
| <b>Railroad Diagram Examples with Sample Input</b> ..... | C-7 |

|                    |   |
|--------------------|---|
| <b>Index</b> ..... | 1 |
|--------------------|---|



# Figures

|      |  |      |
|------|--|------|
| 2-1. | Application Life Cycle .....   | 2-2  |
| 3-1. | Relationships between the Classes in the ORGANIZATION Database ..... | 3-15 |



# Tables

|      |  |      |
|------|--|------|
| 2-1. | Placement of Relationship Attributes .....                       | 2-13 |
| 5-1. | Relationships between Explicitly Declared EVAs .....             | 5-16 |
| 5-2. | Relationships between Declared EVA and Implicit EVA .....        | 5-17 |
| 9-1. | Global Options That Critically Affect Performance .....          | 9-8  |
| 9-2. | Factors Determining the Sets Present for a Multivalued DVA ..... | 9-36 |
| A-1. | SIM Reserved Words .....   | A-1  |
| C-1. | Elements of a Railroad Diagram .....                             | C-2  |





# About This Guide

## Purpose

This guide describes how to use the Object Definition Language (ODL) to define a Semantic Information Manager (SIM) database by specifying the schema directly in a text file. The guide also describes how to use the SIM Utility to process a schema in order to create or maintain a SIM database.

The direct use of ODL is an alternative to using the InfoExec Advanced Data Dictionary System (ADDSS) screen interface or the SIM Design Assistant (SDA) to define a SIM database.

## Scope

Ground rules and guidelines for defining a SIM database are presented. Syntax rules, constraints, and conventions for using ODL are described. Each kind of ODL declaration that you can include in a schema is described. Instructions for creating and maintaining a database are presented.

## Audience

This guide is written primarily for applications and systems developers who are responsible for defining and administering SIM databases.

## Prerequisites

As a user of this guide, you are assumed to have an understanding of the basic concepts of a SIM database.

## How to Use This Guide

When you begin to use this guide, read the first five sections in their entirety. These sections are titled "Introduction to ODL," "SIM Database Design Guidelines," "Basic ODL Considerations," "Types," and "Classes." After that, you can refer again to the information in these sections as needed. Refer to the remaining information in this guide according to your specific needs.

# Organization

This guide contains 12 sections and 3 appendixes, each of which is described in the following text. An index appears at the end of the guide.

### **Section 1. Introduction to ODL**

This section provides an overview of the purpose and nature of ODL. The section also describes SIM database schemas and various techniques for developing them. Finally, the section describes ODL declarations and presents a progressive example of a simple SIM database schema.

### **Section 2. SIM Database Design Guidelines**

This section presents information to help you determine what you need to know before you construct a schema. This information consists of guidelines for designing a SIM database. A recommended three-phase design process is described.

### **Section 3. Basic ODL Considerations**

This section describes the kinds of ODL declarations that you can include in a database schema, and the means and media for creating, amending, or viewing the schema. The section also presents the syntax rules and the language conventions for using ODL. The section also describes the constant data values that are used in ODL declarations. Finally, the section presents an example database called the ORGANIZATION database, which is reflected in examples throughout this guide.

### **Section 4. Types**

This section describes the SIM concept of types and the use of ODL to define and apply types.

### **Section 5. Classes**

This section describes the use of ODL to define classes (base classes and subclasses) and their attributes (class attributes, data-valued attributes (DVAs), and entity-valued attributes (EVAs)) in a database schema, and explains the purposes of classes and attributes.

### **Section 6. Indexes**

This section describes the use of ODL to define indexes in a database schema, and explains the purpose of indexes.

### **Section 7. Verifies**

This section describes the use of ODL to define verifies in a database schema, and explains the purpose of verifies.

### **Section 8. Security**

This section describes the use of ODL to define accesses and permissions in a database schema, and explains the purpose of accesses and permissions. These two kinds of declarations are used together for security purposes.

### **Section 9. DMSII Mapping Options**

This section describes the use of ODL to specify, in a database schema, options that define how a SIM database is mapped onto a DMSII database, and explains the purpose of these DMSII mapping options.

### **Section 10. Dictionary Options**

This section describes the use of ODL to specify, in a database schema, dictionary options that allow the schema to be integrated with the Advanced Data Dictionary System (ADDSD), and explains the purpose of these dictionary options.

### **Section 11. SIM Utility**

This section describes the operation and functions of the SIM Utility with respect to creating and maintaining SIM databases.

### **Section 12. Changing a SIM Database Schema**

This section identifies the kinds of schema changes to a SIM database that are allowed by SIM, and notes special restrictions or implications, where applicable, for allowed changes. The section also identifies the kinds of schema changes that are not allowed by SIM.

### **Appendix A. SIM Reserved Words**

This appendix contains an alphabetic listing of SIM reserved words, namely words that SIM does not allow you to use for identifiers.

### **Appendix B. Syntax for Utility Commands and ODL**

This appendix contains all the railroad diagrams presented in Sections 4 through 11 of this guide. The diagrams show the syntax for the SIM Utility commands relevant to SIM databases and the syntax for ODL.

### **Appendix C. Understanding Railroad Diagrams**

This appendix explains the use of railroad diagrams to describe the ODL syntax in this guide.

# Related Product Information

Unless otherwise stated, all documents referred to in this publication are MCP/AS documents. The titles have been shortened for increased usability and ease of reading.

The following documents are included with the software release documentation and provide general reference information:

- The *Glossary* includes definitions of terms used in this document.
- The *Documentation Road Map* is a pictorial representation of the Product Information (PI) library. You follow paths through the road map based on tasks you want to perform. The paths lead to the documents you need for those tasks. The Road Map is available on the PI Library CD-ROM. If you know what you want to do, but don't know where to find the information, start with the Documentation Road Map.
- The *Information Availability List* (IAL) lists all user documents, online help, and HTML files in the library. The list is sorted by title and by part number.

The following documents provide information that is directly related to the primary subject of this publication.

### ***Data Management Software Installation Guide***

The shortened title for this document is the *Data Management Installation Guide*.

This guide provides procedures for installing data management software for the first time, upgrading data management software to a new release level, and returning data management software to a previous release level. In addition, this guide describes the InfoExec configuration control facility, and provides information on the physical limitations and memory requirements of the database management systems (DBMSs), including Data Management System II (DMSII), Semantic Information Manager (SIM), and Structured Query Language Database (SQLDB). This guide is written for users responsible for installing data management software.

### ***InfoExec Advanced Data Dictionary System (ADDS) Operations Guide***

The shortened title for this document is the *InfoExec ADDS Operations Guide*.

This guide describes InfoExec ADDS operations, such as creating and managing database descriptions. This guide is written for those who collect, organize, define, and maintain data and who are familiar with the Data Management System II (DMSII), the Semantic Information Manager (SIM), and the Structured Query Language Database (SQLDB).

### ***InfoExec Semantic Information Manager (SIM) Object Manipulation Language (OML) Programming Guide***

The shortened title for this document is the *SIM OML Programming Guide*.

This guide describes how to interrogate and update SIM databases using SIM OML. Also described are two methods for processing queries; one method embeds calls on the SIM library in an application program and the other method uses the InfoExec Interactive

Query Facility (IQF). This guide is written for application programmers and experienced IQF and Workstation Query Facility (WQF) users.

### ***InfoExec Semantic Information Manager (SIM) Technical Overview***

The shortened title for this document is the *InfoExec SIM Technical Overview*.

This overview describes the SIM concepts on which the InfoExec data management system is based. This overview is written for end users, applications programmers, database designers, and database administrators.

### ***Personal Workstation<sup>2</sup> InfoExec Semantic Information Manager (SIM) Design Assistant (SDA) Operations Guide***

The shortened title for this document is the *SDA Operations Guide*.

This guide describes the Semantic Information Manager (SIM) Design Assistant (SDA), which is used to design, browse, and maintain SIM database schemas on a personal computer (PC). It contains information on operating SDA, including descriptions and explanations of views, forms, and menus. This guide is written for anyone who designs, updates, or views SIM database schemas.



# Section 1

## Introduction to ODL

The Object Definition Language (ODL) is one of two central languages provided by the Semantic Information Manager (SIM) database system. The other central language is the Object Manipulation Language (OML). OML is used to retrieve, store, and update data in SIM databases, whereas ODL is used to define the structure and behavior of data in SIM databases. The schemas constructed using ODL can be passed to the SIM Utility to create new SIM databases and to change the definition of existing SIM databases.

This section provides an overview of the purpose and nature of ODL. Although you are assumed to be already familiar with basic database concepts, SIM, and the InfoExec system, some background information on these subjects is presented for review purposes. Next, SIM database schemas and various techniques for developing them are described. Finally, ODL declarations are described, and a progressive example of a simple SIM database schema is presented.

## Databases

Many computer applications require the services of a database. A database is a collection of logically related data. The data within a database is persistent because it is retained after the application programs that created it have ended. A database can contain the data for a single application, such as a payroll system, or it can contain the data for a number of related applications, such as a suite of financial applications.

## Database Management Systems

Each database is managed by a database management system (DBMS) that provides shared access to data with some degree of integrity and security. Application programs interact with the DBMS to perform queries that update and retrieve data in the database. The DBMS treats each database as a recoverable unit, ensuring that it is synchronized for consistency throughout all updates and errors that occur to the database.

# Data Models

The components and rules that a DBMS offers for creating and accessing databases constitute its data model. Several data models are described as follows:

- The hierarchical and network data models use a record as the basic unit of data. Records are composed of data-valued items, and they are stored in data sets. A navigational query language is used to store and retrieve records one record at a time.
- The relational data model uses a tuple as the basic unit of data. Tuples are composed of data-valued columns, and they are stored in tables. A set-oriented language (many tuples at a time, usually Structured Query Language (SQL)), is used to store and retrieve tuples.
- The semantic data model (SDM) uses an entity as the basic unit of data. Entities are composed of data-valued or entity-valued attributes, and they are stored in classes. Classes can be arranged in hierarchies that promote such concepts as subclasses, superclasses, and inheritance. A set-oriented query language is used to update and retrieve entities.

The data model employed by a DBMS determines the conceptual manner in which users describe and access data within the database. Physically, databases are composed of a series of files that are managed by the DBMS.

## The Semantic Information Manager (SIM)

SIM is a DBMS that is based primarily on two converging database technologies: SDM and object-oriented databases (OODBs). SIM provides features from each of these technologies as well as additional features. Examples of these features are the following:

- Concepts that are common to both SDM and OODBs, such as entities (or objects), attributes, classes, hierarchies, and inheritance
- Concepts of the structural semantics emphasized in SDM, such as bidirectional relationships, multivalued attributes, and integrity constraints
- Concepts of the behavioral semantics emphasized in OODB technology, such as encapsulated operations and polymorphism
- Such additional features as versatile data types, general integrity constraints, semantic-based security, and OML, which is a, set-oriented data manipulation language that can be used for both programmatic and ad hoc database access

These features give SIM capabilities that DBMSs based on record-oriented or tuple-oriented data models do not have. Increased semantics allow SIM databases to assume more responsibilities for data integrity and security. The high-level data model used by SIM allows SIM databases and applications to more closely pattern real-world business data and to address complex application domains. These features translate to simplified application programs and increased development productivity.



## The InfoExec System

SIM serves as the central component of the InfoExec system. Other InfoExec components include development and support tools such as a data dictionary, a database design facility, host language interfaces, query products, and database maintenance tools. The semantic and object-oriented capabilities of SIM in combination with the high-productivity tools of InfoExec provide a complete environment for database application development, execution, and management.

## Entities and Objects

The primary unit of data in SIM is an entity, which is also referred to as an object. The term *entity* originated in the database field from such data models as SDM. The term *object* originated in the object-oriented programming and OODB fields. Since SIM employs concepts from both the SDM and object-oriented technologies, either term is applicable to SIM. Note that the term *entity* is used throughout this guide for consistency. Note also that the data definition and data manipulation languages of SIM are named ODL and OML as reminders of its object-oriented principles.

An entity is the basic unit of data in SIM because all data is described and accessed in terms of entities. To define a SIM database, for example, you specify the classes in which entities are to be stored and the attributes that they are to possess. When you access a SIM database, you construct OML statements that specify the entities that are to be inserted, modified, deleted, or retrieved.

## Database Schemas

Every DBMS provides a language for defining a database, that is, for defining the structure and behavior of data in a database. Generically, this language is called a data definition language (DDL). The DDL provided specifically by SIM for defining a SIM database is called the Object Definition Language (ODL).

You define a new SIM database by creating a schema that is written in ODL. The schema contains ODL declarations such as the classes and attributes that are desired in the new SIM database.

To change the definition of an existing SIM database, you modify the current schema of the database. In this case, existing ODL declarations might be amended or deleted, and new ODL declarations might be added to reflect the desired database definition.

To implement the new or changed SIM database definition, the schema is directed to the SIM Utility for processing. The SIM Utility is used to create and change all SIM databases.

# Database Schema Development

The InfoExec environment offers the following alternatives with which you can create and manage SIM database schemas and process the SIM Utility:

- Direct schema development
- SIM Design Assistant (SDA)
- Advanced Data Dictionary System (ADDS)

These alternatives are described in the following paragraphs. Each alternative has different advantages, so that you can choose the alternative that meets your particular needs.

## Direct Schema Development

You can directly create or modify a SIM database schema as a text file (of ODL syntax) by using an editing facility such as the Command and Edit language (CANDE) or the Editor program. You then execute the SIM Utility by using any of several methods such as CANDE or a Work Flow Language (WFL) job. When you run the SIM Utility, you must pass the appropriate command, options, and file equations to direct the utility to perform the desired function.

This direct ODL approach requires detailed knowledge of ODL syntax and the SIM Utility program, but it is advantageous for those who prefer a command-oriented approach. The direct ODL approach is the primary focus of this guide. Various sections contain descriptions of the exact syntax needed for each ODL construct and for the operation of the SIM Utility.

## SIM Design Assistant (SDA)

The SDA product enables you to create or modify SIM database schemas on a workstation using a graphical, mouse-supported interface. SDA offers a variety of techniques for viewing, browsing, and editing SIM database schemas. With each function you perform, SDA automatically generates the necessary ODL syntax, and it automatically processes the SIM Utility.

SDA significantly reduces the need for detailed knowledge of ODL syntax and the SIM Utility. It offers many other advantages as well, such as the following:

- Independent workstation development
- Advanced editing features
- Multiple schema handling
- Detailed report generation

SDA represents a high productivity approach to developing and managing SIM database schemas.

For SDA users, this guide provides SIM database design guidelines and necessary information on basic SIM database constructs. Precise knowledge of ODL syntax and the SIM Utility are not needed by SDA users.

## Advanced Data Dictionary System (ADDS)

Regardless of the development method, all SIM database schemas can be loaded into and controlled by ADDS. Once a schema is in ADDS, you can modify it and implement the corresponding changes by using a forms-mode, menu-oriented interface provided by ADDS. Like SDA, ADDS automatically generates the necessary ODL syntax and processes the SIM Utility.

ADDS also allows you to create new schemas and to generate the corresponding SIM database through the menu-oriented interface. As with SDA, ADDS reduces the need for detailed knowledge of ODL syntax and the SIM Utility, although this guide is still relevant for basic information on SIM database constructs.

Depending on how SIM is installed at your site, all SIM database schemas may be required to be loaded into and controlled by ADDS. Specifically, if the option *REPOSITORY = REQUIRED* is specified in the InfoExec SL/CONFIG file at your site, then active ADDS integration is enforced. In this case, you must either define all new SIM databases through ADDS, or specify appropriate dictionary options when defining a new schema through direct schema development or SDA.

If the option *REPOSITORY = REQUIRED* is not in effect at your site, you can create SIM databases through direct schema development or SDA without integrating them with ADDS. You can load such databases into ADDS at any time by adding an appropriate dictionary-options declaration to the schema and processing the corresponding changes.

Once a SIM database schema is loaded into ADDS, security specified by ADDS determines who can inquire upon or modify the schema. Furthermore, security is automatically enforced by the SIM Utility regardless of which method is used to access the schema. Therefore, ADDS can be used to provide automated security and control over SIM database schemas.

Besides schema security, there are other advantages of having SIM database schemas controlled by ADDS, such as the following:

- Centralized administration
- Version control
- Report generation

Refer to Section 10, "Dictionary Options," for more information on dictionary options and the integration of SIM databases with ADDS. Refer to the *InfoExec ADDS Operations Guide* for more information on the use of ADDS.

# ODL Declarations

ODL is a declarative language, which means that ODL consists entirely of declarations. Unlike programming languages, ODL includes no executable statements.

## Kinds of ODL Declarations

The kinds of ODL declarations that you can specify when you define a SIM database schema are the following:

- User-defined type
- Base class
- Subclass
- Index
- Verify
- Access
- Permission
- DMSII mapping options
- Dictionary options

These ODL declarations are described briefly in Section 3, “Basic ODL Considerations,” and they are described in detail in subsequent sections.

## Categories of ODL Constructs

ODL declarations consist of constructs that can be divided into three categories:

- Some ODL constructs define basic database structure. These constructs determine the types of entities that the database is to maintain and the information that is to be kept by each entity. Constructs in this category include base classes, subclasses, and attributes.
- Some ODL constructs define additional semantics beyond those implied by the basic database structure. These constructs increase the data integrity and the security of data by giving the database more information about the desired behavior of data. Constructs in this category include user-defined types, attribute options, verifies, accesses, and permissions.
- Some ODL constructs define implementation options that are in addition to or alternatives to the defaults chosen by SIM. These constructs can be used to increase database performance or to specify operational characteristics of a SIM database. Constructs in this category include indexes, DMSII options, and dictionary options.

## Example Database Schema Development

To illustrate basic ODL declarations and their constructs, an incremental example is presented in the following paragraphs. This example shows, in a stepwise manner, the aforementioned three categories of ODL constructs being introduced into a SIM database schema. Since the example is presented for general illustrative purposes, each ODL construct that it shows is not fully explained. Examples presented later in this guide provide more complete explanations of the constructs they show.

Assume that a simple project management application is being developed for a business. For this application, you are given the task of defining a SIM database to keep track of current employees and the projects that they work on. From this task description, two kinds of entities for the database are recognized: employees and projects. Furthermore, it is recognized that each employee has certain characteristics such as a name, an employee number, and a manager, who is also an employee. It is also recognized that each project has a project number, a title, and a team of employees. With this information, you have identified the basic structure of the database, which you can express with the ODL declarations in the following schema:

```
CLASS Employee
  (name          : STRING[20];
   employee-id   : INTEGER;
   employee-manager : Employee;
  );

CLASS Project
  (project-no      : INTEGER;
   project-title   : STRING[20];
   project-members : Employee, MV;
  );
```

So far, you have defined a schema that contains two classes, Employee and Project, to hold the two kinds of entities that have been identified. Each class has three attributes that correspond to the entity characteristics that have been identified.

Next, suppose that you wish to increase the semantics that are known to and controlled by the database. It is determined that every employee must have a name, that an employee-id can have only unique, positive values between 1 and 99999, and that an employee cannot be his or her own manager. Also, it is determined that each project must have either a project-no or a project-title or both, that a project-no must have a value between 1 and 10000 or the special value 99999, and that a project can have no more than 15 team members. Moreover, it is determined that this database is to be accessible only to users whose usercode is PROJECT, and that such users are to have full access to the database.

You can enforce these semantics by amending the schema as follows (amended portions are underscored):

```
CLASS Employee
  (name                : STRING[20] REQUIRED;
   _____
   employee-id         : INTEGER (1 .. 99999) UNIQUE;
   _____
   employee-manager    : Employee;
  );

VERIFY Valid-manager ON Employee:
-----
employee-manager NEQ Employee
-----
ELSE "An employee cannot be his or her own manager";
-----

CLASS Project
  (project-no          : INTEGER (1 .. 10000, 99999);
   _____
   project-title       : STRING[20];
   project-members     : Employee, MV (MAX 15);
   _____
  );

VERIFY Valid-project-identification ON Project:
-----
project-no EXISTS OR project-title EXISTS
-----
ELSE "A project must have a project-no or a project-title";
-----

ACCESS Database-access ON ALLDB
-----
ALLDML;
-----

PERMISSION Database-permission
-----
USERCODE = (PROJECTS),
-----
ACCESS   = Database-access;
-----
```

Finally, suppose that you wish to improve database performance by adding some options. For example, it is desired to provide efficient access to employees when searched by name. Also, since the employee-manager attribute represents a one-to-many relationship, it is desired to implement the relationship by using the foreign key mapping technique instead of the default mapping.

You can enforce these requirements by amending the schema as follows (amended portions are underscored):

```
CLASS Employee
  (name           : STRING[20] REQUIRED;
   employee-id    : INTEGER (1 .. 99999) UNIQUE;
   employee-manager : Employee;
  );

VERIFY Valid-manager ON Employee:
  employee-manager NEQ Employee:
  ELSE "An employee cannot be his or her own manager";

INDEX Employee-by-name ON Employee
-----
  (name);
-----

CLASS Project
  (project-no      : INTEGER (1 .. 10000, 99999);
   project-title   : STRING[20];
   project-members : Employee, MV (MAX 15);
  );

VERIFY Valid-project-identification ON Project:
  project-no EXISTS OR project-title EXISTS
  ELSE "A project must have a project-no or a project-title";

ACCESS Database-access ON ALLDB
  ALLDML;

PERMISSION Database-permission
  USERCODE = (PROJECTS),
  ACCESS   = Database-access;

DMSII-OPTIONS
-----
  (CLASS Employee
   -----
     (ATTRIBUTE employee-manager (MAPPING = FOREIGN-KEY));
   -----
  );
--
```

Note that the preceding example is a simplified version of the ORGANIZATION database schema that is used in examples throughout this guide. The complete ORGANIZATION database schema is presented in Section 3, "Basic ODL Considerations."





# Section 2

## SIM Database Design Guidelines

This section presents guidelines for designing a SIM database. These guidelines are applicable whether you define a SIM database schema using the direct ODL approach, SDA, or ADDS. They are intended to help you take full advantage of the unique capabilities of SIM and to help you maximize the completeness and effectiveness of your database. While these guidelines do not constitute a complete development methodology, they do propose a basic process for designing a SIM database.

You can read this section to gain familiarity with general SIM concepts, the SIM data model, and ODL. However, you should be fully familiar with these concepts when you are ready to design a new SIM database.

The information in this section is presented under three general headings:

- **Initial Considerations**  
Some initial topics are discussed that you should consider before beginning the design of a SIM database. These topics include the application life cycle, development methodologies, and prototyping.
- **SIM Database Design**  
A process is proposed that you can use to design a SIM database. This process consists of a series of specific phases and steps, each of which addresses a distinct design objective. Numerous suggestions are included for evaluating and refining a SIM database schema.
- **The SIM Data Model beyond Databases**  
The applicability of the SIM data model to other application components is discussed. This information suggests how you can further exploit the modeling capabilities of SIM and integrate overall application development.

### Initial Considerations

A database constitutes only a single component of a complete application system. Systems that employ a database typically require other software components, such as transaction processing programs, user interface modules, and data communications modules. In addition, complete application systems usually require nonsoftware components, such as documentation and training courses. Furthermore, the initial development of a system represents only a fragment of the overall life span of the system. Application systems are typically enhanced and maintained for many years after their first release.

To ensure initial and continuing success of your application system, you should consider formal techniques for developing individual components and for managing the system as a whole. Some topics relative to overall system development are discussed in the following paragraphs.

### The Application Life Cycle

The process by which application systems are conceived, developed, installed, and maintained is called the application life cycle. Many models have been produced to characterize the life cycle and to propose techniques for managing them. Most life-cycle models consist of a series of iterative phases, such as those depicted in Figure 2-1.

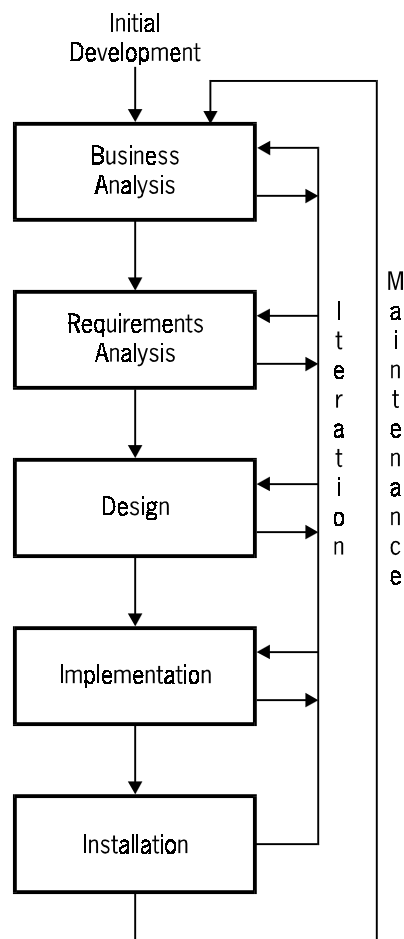


Figure 2-1. Application Life Cycle

The purpose of each phase within this model of the application life cycle is summarized as follows:

- **Business analysis**

The purpose of this phase is to identify and define the business objectives and needs of a company, an organization, or a specific operation. A document is produced, such as a business plan. Business operations are specified in such areas as mission-critical (central to the core business), administrative (day-to-day monitoring and control), and executive (long-term forecasting, planning, and direction).

- **Requirements analysis**

The purpose of this phase is to identify and study a business problem that warrants a computerized solution. Solution requirements are decomposed and organized. Detailed requirements are specified in a document such as a requirements specification.

- **Design**

The purpose of this phase is to produce a solution implementation plan. An application system is proposed and detailed in a document such as a functional or detailed design. More than one potential solution may be defined, each with a projected feasibility, cost, and schedule. Design documentation must address many topics, including implementation strategy, test plans, integration with existing applications, product documentation, installation and release plans, and so on.

- **Implementation**

The purpose of this phase is to execute the implementation plan to produce the completed application system. Code is written, tests are performed, adherence to design and requirements is verified, parallel tests may be run with existing systems, and so on. The system is prepared for delivery.

- **Installation**

The purpose of this phase is to deliver and install the implemented solution. Users are trained, old systems are migrated or phased out, and the new system is activated.

This life-cycle model is not necessarily comprehensive, but it does reflect a typical application development approach. Individual development phases are rarely performed in an entirely sequential manner. Instead, the entire development process is actually iterative: work being done at any phase might detect deficiencies that prompt a return to a previous phase. Furthermore, different application components and different developers can be in different phases at the same time.

At some point in the development life cycle, the threshold that separates problem statement from solution statement is crossed. This crossover to solution statement usually occurs in the design phase, but it sometimes occurs earlier. It is important to keep track of problem elements (requirements) and solution elements (implementation) separately so that individual elements can be described and verified by appropriate personnel (such as users, programmers, and database designers).

Once the system is installed, it becomes actively used (sometimes called “in production”). At that point, the system is monitored for performance and problems. The identification of new requirements causes the system to go into a maintenance cycle in which the life cycle is repeated: new requirements are analyzed, designed, implemented, and installed. The majority of software costs come not from initial application development but from application maintenance.

When you develop an application system, you should employ some kind of formal process to perform each development phase and to manage the overall application life cycle. Such a process can help to ensure the accuracy, completeness, and quality of your system. Within the process, you can use any of several techniques to develop and maintain specific application components. These techniques are termed *methodologies*.

## Development Methodologies

Within the application life cycle, various methodologies can be used to develop specific components such as programs and databases. A methodology must be appropriate for the type of component being built, and it must integrate the development of separate components within the application. Some common methodologies for program and database development are described in the following paragraphs.

### Program Development

For program development, a common methodology is structured analysis and design. This methodology uses a mixture of data-driven and process-driven techniques for transforming business requirements into an implemented system. Various representation techniques are used to describe requirements, data analysis, and task hierarchies. One drawback of structured methodologies is that they do not work well for databases.

Another program development methodology that is coming into more widespread use is object-oriented analysis and design. This methodology uses techniques that integrate the structure and behavior of problem and solution elements into objects. Object-oriented methodologies are especially well-suited for programs that interface with SIM databases because the SIM data model—and the database design process described in this section—are based, in part, on object-oriented concepts.

### Database Development

A common methodology for database development is entity-relationship (E-R) modeling. This methodology uses the entity-relationship data model and a process that arrives at a conceptual schema representing the desired database design. However, since most commercial databases use a record-based or tuple-based data model, the E-R conceptual schema must be translated into the language of another database model, such as a relational model. This process is usually manual, and it requires complex steps such as normalization. Furthermore, the translation causes the database to lose semantics, which then must be captured by programs or people.

### SIM Database Development

When you develop a SIM database, you can use E-R modeling and directly translate an E-R conceptual schema into a SIM database schema without losing semantics. You can do so because SIM directly supports basic E-R concepts such as entities, relationships, and attributes. However, the SIM data model is somewhat more advanced than the E-R data model since the SIM data model supports extended concepts such as bidirectional relationships, generalization hierarchies, and general integrity constraints. Therefore, you can more optimally develop a SIM database schema by directly incorporating its data model.

The database design process that is presented in this section is a set of techniques that you can use to develop a SIM database. The process incorporates elements from E-R modeling, object-oriented techniques, and the semantic data model. Although this process does not constitute a complete methodology (which would require a book in itself), it should help you to achieve a SIM database that meets the business requirements of your application and that reflects an effective use of SIM.

Before the database design process is presented, the subject of prototyping is briefly discussed.

### Prototyping

Prototyping is the fast development and testing of a small-scale version of an application component or system. A prototype typically contains realistic core implementation elements, but many noncore features are omitted or simplified. The objective of prototyping is to validate a specific design approach early yet accurately. A prototype should validate all basic aspects of a design, including architecture, interfaces, and physical considerations such as capacity and performance.

The use of prototyping during the development process has many advantages. Prototypes serve as a user feedback mechanism to verify requirements and to foster details. Also, prototypes help you find problems early—when they are cheaper to correct. Finally, prototypes do not have to be discarded: if carefully constructed, a prototype can serve as the initial application system and can simply be expanded to achieve full implementation.

SIM is especially well-suited for prototyping because you can quickly develop a SIM database schema and implement the corresponding database. Furthermore, the InfoExec environment provides tools that enable you to quickly construct and execute OML queries so that behavior, performance, capacity, and other aspects of a SIM database can be studied.

Consequently, it is highly recommended that you use prototyping while designing a SIM database. A suggested approach is to use the SIM Design Assistant (SDA) product to construct and implement SIM database schemas, and to use either the Interactive Query Facility (IQF), the Workstation Query Facility (WQF), or the programmatic query interface to define and execute OML queries.

# SIM Database Design

A process that you can use to design a SIM database is presented in the following paragraphs. You should consider this design process as a set of guidelines since individual databases must be designed with particular application and development requirements in mind. First, an overview of the overall design process is presented.

## Overview

You can begin the design of a SIM database in either the requirements analysis phase or the design phase of your application system. Many application requirements translate directly into SIM database constructs. Consequently, a SIM database schema can be useful for documentation and feedback in the early phases of development. For this reason, it is recommended that you begin SIM database design as early as possible. Note that when you develop a SIM database schema as a requirements task, you can still consider the process as database design since the schema is a specification that can be directly implemented.

The recommended design process is divided into three phases that you should perform independently. These phases translate certain application requirements into SIM database constructs that are elaborated and refined. The three design phases are summarized as follows:

- Logical design

The purpose of this design phase is to develop the logical database schema. This phase concentrates on the identification of persistent data, the definition of logical database structure and semantics, and adherence to business rules and application requirements. The logical design phase focuses on database structure and semantics as described in Section 1, "Introduction to ODL."

- Interface design

The purpose of this phase is to determine how the database is to be used by application programs and users. This phase identifies the types of queries and transactions that are to be performed against the database, and it identifies any additional information that is necessary to facilitate these accesses. The interface design phase refines database structure and semantics to ensure a proper database "fit" with other application components.

- Physical design

The purpose of this phase is to determine how the logical schema should be mapped to a physical schema. This phase considers such criteria as capacity requirements, performance requirements, operational requirements, and schema evolution requirements. The physical design phase consists of adding and refining options as described in Section 1.

Each design phase consists of a series of steps that accomplish specific objectives. When you begin a new SIM database design, you should perform these phases in the order shown. However, as in the life-cycle phases, you will probably want to repeat these phases and the steps within them several times to achieve an optimal database schema. Also, if several developers are contributing to the database design, different areas of the database may be in different phases at the same time.

Throughout this discussion, the ORGANIZATION database is used to illustrate various suggestions. Examples include ODL declarations that represent a portion of the ORGANIZATION database schema. A description of the ORGANIZATION database and its complete ODL schema is presented in Section 3, “Basic ODL Considerations.”

## Logical Design

The goal of the logical design phase is to establish the structure and semantics of the database. This aspect of the schema is sometimes referred to as the conceptual schema. As described in Section 1, “Introduction to ODL,” database structure includes such constructs as classes, subclasses, and attributes. Database semantics include such constructs as integrity constraints and security. The logical design phase is divided into the following steps:

- Step 1: Identify things and their properties.
- Step 2: Generalize and specialize.
- Step 3: Add semantics.
- Step 4: Evaluate and refine.

For your first pass through the logical design phase, you should perform these steps in order. Undoubtedly, you will want to iterate through these steps several times in order to achieve a complete logical database design. Each step is described in the following paragraphs.

### Step 1: Identify Things and Their Properties

The basic purpose of a database is to store the persistent data required by an application system. In a SIM database, this data must be defined in terms of the basic unit of data for SIM: the entity. The purpose of this step is to identify the types of entities that the database is to manage. In simple terms, you will identify “things” to be stored by the database and the properties of these things.

Before you begin this step, you should be familiar with the general concepts of classes, attributes, and class attributes. These constructs are described in Section 5, “Classes,” although some reminders and tips are given as this step is described.

### Identifying Classes

Anything can be an entity. Entities can be very concrete (such as a person or a telephone) or very abstract (such as an event, a time period, or a relationship instance). As you identify entities, disregard physical considerations, such as the manner in which entities are mapped into records. (If you are accustomed to record-based data models, breaking away from record orientation is one of the most important yet difficult things to do.)

Actually, you do not define individual entities in a schema. You define entity types by declaring the classes in which entities are to be stored. For each type of entity that you identify, you should declare one class. Each class must have a unique name, such as Project or Employee.

### Identifying Attributes

When you declare a class, you should begin to define the attributes that correspond to the entities in that class. Use the following guidelines when you define attributes:

- Attributes are the properties that each entity of a class can possess. An attribute is a characteristic such as an identification (name, number), a feature (price, quantity on hand), or a relationship (manager, components, suppliers).
- Each attribute must have a name (such as project-no or manager) and a type, which indicates the range of values that the attribute can hold.
- If the type for an attribute is data-valued, such as INTEGER, STRING [20], or DATE, the attribute is a data-valued attribute (DVA). DVAs hold printable data values.
- If the type for an attribute is another class, such as Project or Employee, it is an entity-valued attribute (EVA). EVAs hold references to other entities.
- After its name and type, the next most important consideration for an attribute is its cardinality. If each entity in the class can have only one value for an attribute (as in project-no), the attribute is single valued (SV), which is the default. If the attribute can have two or more values for each entity (as in children), the attribute is multivalued (MV). Specify MV if an attribute can have multiple values; otherwise let it default to SV.
- If attribute options other than cardinality (such as REQUIRED, UNIQUE, MAX, or a subrange) are obvious when you define an attribute, define them. Otherwise, do not be concerned about these attribute options in this design step; you can focus on them later.



In addition to attribute options, there are other constructs that you can specify if they are obvious. However you should not work too hard to define these constructs in this design step; they can be addressed in later design steps. Examples of such constructs are the following:

- Hierarchies

Do not struggle to arrange classes into precise hierarchies by defining certain classes as subclasses. If something is an obvious subclass of another class, go ahead and specify it as such. Otherwise, make each class a base class.

- Additional semantics

These semantics include user-defined types, verifies, and security. If any of these semantics are obvious, add them. Otherwise, they can be addressed later.

- Interface considerations

These considerations include program and user considerations, such as data formats and query usage.

You can specify attribute options and constructs in the schema if they are obvious. However, you should avoid addressing physical considerations even when they are obvious. These physical considerations include such concerns as physical mapping, performance, and capacity. The logical database design must be correct before these physical concerns are considered, and you could needlessly distort the logical schema if you attempt to address physical concerns too early.

### **Example**

The following example illustrates the guidelines given in step 1 so far.

Assume that you are developing an ORGANIZATION database that is an extension to the project management database described in Section 1, "Introduction to ODL." Besides the Employee and Project classes and their attributes as identified in Section 1, assume that you also identify the need to keep track of employees' children, John Smith (the president of the company), and previous employees of the company.

Since each employee can have more than one child, children is a multivalued attribute. Because John Smith is an instance of some type (even if he appears to be the only instance), you should invent a class such as President to house this entity. Similarly, previous employees represent a new entity type, so you should invent a class for it. The corresponding ODL declarations are shown in the following schema:

```
CLASS Employee      "Current employees of the company"
  (name              "Employee's first and last name"
   : STRING [20];
   employee-id       "Uniquely identifies each employee"
   : INTEGER;
   employee-manager   "Employee's current manager"
   : Employee;
   children           "Employee's children"
   : STRING [20], MV;
  );

CLASS Project        "Current projects in progress"
  (project-no        "Uniquely identifies each project"
   : INTEGER;
   project-title      "Code name for project"
   : STRING [20];
   project-members    "Current employees on project"
   : Employee, MV;
  );

CLASS President      "The current president of the company"
  (name              "President's first and last name"
   : STRING [20];
  );

CLASS Previous-Employee "Past employees of the company"
  (name              "Past employee's name"
   : STRING [20];
  );
```

As shown, this schema concentrates on capturing types of things (classes) and their properties (attributes). There is potential redundancy, generalization, and other refinements that need to be addressed, but these are not concerns at the moment. The schema does show how an *official comment* can be used to document each declaration. An official comment is a quoted string that immediately follows the name of a declaration. You should use this feature liberally to describe each declaration's purpose, origin, owner, or any other useful information.

Some suggestions on specific topics within logical design step 1 are presented in the following paragraphs.

### Attribute Placement

Attach each attribute to the class that most clearly “owns” it. For example, project-no is more accurately a property of projects than of employees. Some attributes can be temporarily repeated if necessary. For example, Employee, President, and Previous-Employee each can be given a name attribute. Later, this replication can be scrutinized to determine if it should be eliminated.

### Class Attributes

An attribute has a value (or a set of values) for each entity in a specific class. However, a class attribute, although it still pertains to a specific class, has one value for the entire database. For example, every employee may have a salary, but suppose there is a maximum salary that limits the salary an employee is allowed to have. Because the maximum salary exists only once for all employees, it is a class attribute. These two attributes can be expressed in the schema as follows:

```
CLASS Employee          "Current employees of the company"
  CLASS-ATTRIBUTES
    (maximum-salary     "Maximum salary any employee can have"
      : NUMBER [8, 2];
    )
    (salary              "Salary of employee"
      : NUMBER [8, 2];
    );
```

Incidentally, since maximum-salary obviously serves to fulfill an integrity constraint, you can immediately add the constraint to the schema. For example, you can add the following verify declaration to the schema:

```
VERIFY Valid-employee-salary ON Employee:
  salary LEQ maximum-salary
  ELSE "Employee salary cannot exceed the maximum allowed";
```

### Naming Conventions

The following suggestions for choosing class and attribute names should especially assist programmers and users who plan to construct OML queries against the database:

- Name classes singularly (for example, Project, Employee, and Department) to emphasize that attributes are properties of each entity in the class.
- Name single-valued attributes (single-valued DVAs, single-valued EVAs, and class attributes) singularly (for example, name, dept-assigned, and maximum-salary) since each of these attributes can possess only one value per entity.

- Name multivalued attributes plurally (for example, project-members and children) to reflect the fact that these attributes can have several values per entity.
- If the type for an attribute is not obvious, it may be helpful to choose a name that suggests the type, especially for EVAs (for example, dept-assigned, project-manager).
- Remember that SIM has naming rules for each declaration type, as defined in Section 3, “Basic ODL Considerations.”

### Attribute Types

For DVAs and class attributes, be sure to make full use of the SIM-provided types (especially DATE, TIME, and SYMBOLIC) because these types communicate more semantics. The type for an EVA must correspond to a class. Therefore, if you add an EVA and no existing class sufficiently represents its type, define a new class.

A common dilemma is whether an attribute should be a DVA or an EVA to a new class. For example, suppose you define a Manager class in which each manager has a department to which he or she is assigned. If all that must be stored about a manager’s department is its name, manager-dept can be a DVA whose type is STRING. However, if departments have at least one additional attribute such as a department number, or if two or more classes must reference departments, manager-dept should be an EVA whose type is a Department class. The corresponding ODL syntax is as follows:

```
CLASS Manager          "Managers of the company"
  (manager-dept        "Department to which manager belongs"
    : Department;
  );

CLASS Department       "Departments within the company"
  (dept-name           "Corporate department name"
    : STRING [20];
  dept-no              "Corporate department number"
    : INTEGER;
  );
```

### Relationships

In SIM, every EVA establishes a bidirectional relationship: each EVA has an inverse EVA that describes the same relationship from the opposite direction. An inverse EVA does not have to be explicitly named—SIM declares an implicit inverse. However, there are several advantages to explicitly naming inverses. Some advantages are described in this section and others are described in Section 5, “Classes.”

Sometimes you can identify attributes that appear to belong to a relationship instead of a class. There are several ways to place these “relationship attributes.” A suggested approach is as follows. First, declare the EVA at both ends of the relationship, by using the INVERSE option for at least one of the EVAs, to determine the cardinality of each EVA. Then, look up the cardinality of the two EVAs in Table 2–1 to determine where to place relationship attributes; note that the abbreviations SV and MV denote single valued and multivalued, respectively.

Table 2-1. Placement of Relationship Attributes

| EVA #1<br>Cardinality | EVA #2<br>Cardinality | Relationship<br>Cardinality | Where to Place<br>Relationship Attributes     |
|-----------------------|-----------------------|-----------------------------|---|
| SV                    | SV                    | One-to-one                  | In the class of either EVA                    |
| MV                    | SV                    | One-to-many                 | In the class of the SV EVA                    |
| MV                    | MV                    | Many-to-many                | In a new class "between"<br>the existing EVAs |

For example, suppose that when a manager is assigned to a department, you need to keep track of when the assignment is made. This date-assigned attribute appears to belong to the relationship instead of to one class or the other. First, declare both EVAs in the relationship, which might result in the following schema:

```

CLASS Manager          "Managers of the company"
  (manager-dept        "Department to which manager belongs"
    : Department, INVERSE IS dept-managers;
  );

CLASS Department       "Departments within the company"
  (dept-managers        "Managers for this department"
    : Manager, MV;
  );

```

The relationship cardinality is one-to-many, and, as suggested by Table 2-1, the date-assigned attribute can be declared in the class containing the single-valued EVA, namely Manager.

If manager-dept is multivalued, thereby forming a many-to-many relationship, Table 2-1 suggests that a new class (say Department-Assignment) should be declared "between" the existing EVAs and that the relationship attribute can be declared there.

These declarations are shown in the following schema:

```

CLASS Manager          "Managers of the company"
  (dept-assignments    "Department assignments of Manager"
    : Department-Assignment, MV,
    INVERSE IS assignment-manager;
  );

CLASS Department-Assignment "Manager/Department assignments"
  (assignment-manager   "Manager for this assignment"
    : Manager, REQUIRED;
    assignment-department "Department for this assignment"
    : Department, REQUIRED;
    date-assigned       "Date assignment was made"
    : DATE;
  );

```

```
CLASS Department
  (mgr-assignments
    "Departments within the company"
    "Manager assignments for department"
    : Department-Assignment, MV,
    INVERSE IS assignment-department;
  );
```

Note that the Department-Assignment class causes the many-to-many relationship to become two one-to-many relationships. Also, since each Department-Assignment instance is actually a relationship instance, it must have values for its two EVAs. Consequently, you can immediately specify that its two EVAs are REQUIRED, as shown in the schema.

The use of classes to represent relationship instances is a natural use of the SIM data model: anything can be modeled as an entity. Also, since a class can possess many EVAs, it can be used to model “n-ary” relationships in which more than two entities are involved in each relationship instance.

### Step 2: Generalize and Specialize

The purpose of this logical design step is to organize the classes that you defined in step 1 into generalization hierarchies. Forming hierarchies communicates a high degree of semantic information to SIM, and it also results in a more complete and accurate schema. For example, overlapping and redundant classes are simplified or eliminated, redundant attributes are eliminated, and misplaced attributes are moved to the correct classes. Forming hierarchies draws your attention to entity *roles*. You can create new classes that represent more general roles and new classes that represent more specialized roles.

Before you begin this step, be sure you are familiar with the semantics described in Section 5, “Classes,” under “Generalization Hierarchies.” Be especially sure to understand the concepts of attribute inheritance, least common ancestors (LCA), directed acyclic graphs (DAGs), and the purpose of the subrole data-valued attribute (subrole DVA).

#### Forming Hierarchies

To form hierarchies, you analyze each class to determine whether it is a superclass, a subclass, or a “sibling” of other classes in the schema. Siblings are subclasses that share at least one common ancestor (superclass).

If you determine that a class represents a more general role of some other class, you make it a superclass of that class.

If you determine that a class is a more specific role of some other class, you make it a subclass of that other class.

If you determine that two classes are siblings, you make them subclasses of a common superclass.

Some guidelines that you can use in performing this process for forming hierarchies are presented in the following paragraphs.

### Critical Examples

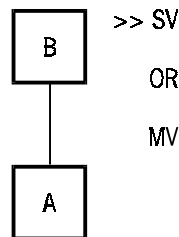
It is often very useful to identify entities that “typify” a class when you are considering the properties of the class and its relationship to other classes. For example, think of typical employees, projects, managers, and so forth when comparing the corresponding classes to other classes. (In a later design step, you will try to find atypical entities that represent extreme conditions or exceptions to the rule.)

### Role Relationships

You can use the following rules to compare two classes, *A* and *B*, and determine whether there is a hierarchical relationship between them. For each rule, the hierarchical relationship corresponding to that rule is graphically illustrated. When a superclass is present, the cardinality of the subrole DVA for that superclass is indicated in the illustration (SV for single valued or MV for multivalued), preceded by the >> symbol.

#### Rule 1

All *As* are *Bs*, but not all *Bs* are *As*: *A* is a subclass of *B*. The subrole DVA of *B* may be single valued or multivalued.



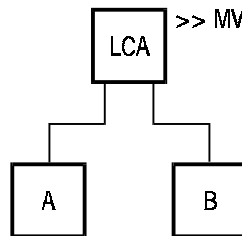
#### Rule 2

All *As* are *Bs* and all *Bs* are *As*: *A* and *B* are probably the same class. For example, if all managers are project leaders and vice versa, there may be no important difference between them. Technically, one might be the subclass of the other, but if all entities that participate in one must participate in the other, the distinction is insignificant, and the two classes should be merged into a single class.



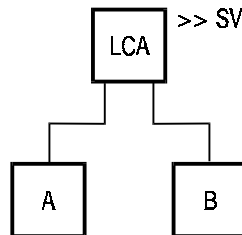
### Rule 3

Some *As* are *Bs* and some *Bs* are *As*: *A* and *B* are siblings. That is, they are subclasses that share at least one common ancestor (superclass). Their least common ancestor (LCA) could be one level up or several levels up, but its subrole DVA is multivalued. In this case, you should identify the LCA class or define a new class that serves as one.



### Rule 4

No *As* are *Bs* and no *Bs* are *As*, but some class *X* exists such that some *Xs* are *As* and other *Xs* are *Bs*: *A* and *B* are siblings whose LCA has a single-valued SUBROLE attribute.



### Rule 5

No *As* are *Bs* and no *Bs* are *As*, and no class *X* exists such that some *Xs* are *As* and other *Xs* are *Bs*: *A* and *B* are in separate hierarchies. That is, they have no common ancestor.



## Redundant Data

Each fact (class or attribute) should be represented only once. Redundant data sometimes indicates that an attribute has merely been duplicated. For example, if Manager and Department both have a department number, delete the attribute from Manager and, if does not already exist, add an EVA between Manager and Department.



In some cases, redundant data might indicate the need for adjusting hierarchies. For example, consider the fact that Employee, Previous-Employee, and President all have names. Sometimes two attributes with the same identifier are not redundant because they represent different types of data (for example, name of Employee versus name of Department). However, with these classes, there is some redundancy that you can resolve as follows:

- Since employees, previous employees, and presidents are all people, their name attributes represent the same type of data. You should resolve the redundancy problem by defining a common superclass, such as Person, that contains the name attribute and then deleting the name attribute from the Employee, Previous-Employee, and President classes.
- The Manager class shown in previous examples is also a subclass of Person. Furthermore, all managers are employees, and all presidents are managers. Consequently, Manager should be a subclass of Employee and President should be a subclass of Manager.
- The children attribute of the Employee class also contains the names of people (a more subtle form of redundancy). Consequently, children should be changed from a multivalued DVA to a multivalued EVA whose type is Person.

Furthermore, both employees and previous employees can have children, so this attribute should be moved up to Person.

- When you define Person, you need to add a subrole DVA, such as employee-status, that defines Employee and Previous-Employee. Since a person cannot fulfill both of these roles at once, the subrole DVA should be single valued. Since some persons may be neither employees nor previous employees (for example, children), the subrole DVA cannot be REQUIRED.
- Similarly, the classes Employee and Manager need a subrole DVA since each of these classes has a subclass.

The hierarchy that results from these refinements is shown in the following schema:

```

CLASS Person                                "Persons related to the company"
  (employee-status                          "Whether person is a current or "
                                           "previous employee of the company"
                                           : SUBROLE (Employee, Previous-Employee);
  name                                     "Person's first and last name"
                                           : STRING [20];
  children                                "Person's children"
                                           : Person, MV;
);

SUBCLASS Employee                          "Current employees of the company"
  OF Person
  (manager-status                          "Whether or not employee is a Manager"
                                           : SUBROLE (Manager);
  employee-id                             "Uniquely identifies each employee"
                                           : INTEGER;
  employee-manager                        "Employee's current manager"
                                           : Manager;

```

```
);

SUBCLASS Manager      "Managers of the company"
  OF Employee
  (president-status    "Whether or not manager is a President"
    : SUBROLE (President);
    manager-dept       "Department to which manager belongs"
    : Department;
  );

SUBCLASS President     "The current president of the company"
  OF Manager;

SUBCLASS Previous-Employee "Past employees of the company"
  OF Person;
```

Some redundancy may be very subtle. For example, if an employee's department is always the same as his or her manager's department, there is no need to have both an employee-dept attribute of Employee and a manager-dept attribute of Manager. One of these EVAs should be deleted, probably the one that has more duplicate values (in this case, employee-dept). However, if an employee's employee-dept can be different for at least one entity than his or her manager's manager-dept, both EVAs should be declared.

### Step 3: Add Semantics

The purpose of this logical design step is to add integrity and security constraints to the database structure that you created in steps 1 and 2. The corresponding increase in semantics has such important benefits as the following:

- Additional application requirements are captured and expressed in the schema, which increases the usefulness of the schema as a design document that can be verified by developers and even users.
- More implementation responsibilities are entrusted to the database, which improves the reliability of the data and reduces application program responsibilities.
- Increased semantics enable SIM to more optimally map the logical schema into a physical schema and to perform useful optimization when executing OML queries.

To perform step 3, you should be familiar with the following:

- Attribute options, which are described in Section 5, "Classes"
- Data types and user-defined types, which are described in Section 4, "Types"
- Indexes, which are described in Section 6, "Indexes"
- Verifies, which are described in Section 7, "Verifies"
- Accesses and permissions, which are described in Section 8, "Security"

## Adding Constraints

To add semantics to your schema, you should examine each class and attribute and look for integrity and security constraints that are not yet reflected. This process will mostly prompt you to add new declarations to the schema, such as attribute options and verifies, but it may also identify the need for further structural refinements to eliminate redundant data, refine hierarchies, and so on. At this point, it is better that you over specify constraints rather than under specify them. Later, semantics that are not sufficiently important or that are too performance-sensitive can be removed.

The ODL constructs that you should use to increase database semantics and the manner in which they should be used are discussed in the following paragraphs.

## Attribute Options

An attribute can have a variety of options, depending on whether it is a DVA or an EVA and on whether it is single valued or multivalued. For each attribute in the schema, consider each of the following options that is applicable:

- SV versus MV

Remember that without either specification, an attribute is considered single valued (SV). If the attribute can hold two or more values for a single entity, make the attribute multivalued (MV). Multivalued EVAs are very common. However, multivalued DVAs are rare because it is usually preferable to declare a multivalued EVA to a new class than to declare a multivalued DVA.

- REQUIRED

An attribute should be declared as REQUIRED unless it is legitimately optional, such as children, or occasionally unknowable, such as birthplace. Attributes that cannot be declared as REQUIRED should be scrutinized as described later in “Step 4: Evaluate and Refine.” Note that the use of the REQUIRED option for EVAs has some restrictions that are described in Section 5, “Classes.”

- UNIQUE

If an attribute value can occur only once for all entities within a class, declare it as UNIQUE. In general, you only need to use the UNIQUE option for DVAs; uniqueness for an EVA is generally implied by the cardinality of its inverse EVA. Furthermore, DVAs that are UNIQUE usually represent identification characteristics, such as name or ID number. Note that UNIQUE does not imply REQUIRED—a UNIQUE attribute can still be null for an arbitrary number of entities in the class.

- INITIALVALUE (DVAs only)

When an entity is inserted into the database, there may be a default value that should be used for one or more of its DVAs. For example, new employees may have a default salary if no explicit salary is assigned. The INITIALVALUE option is particularly useful for satisfying the REQUIRED option when an entity is inserted.

- **DISTINCT (MV attributes only)**

If a multivalued DVA should not have any duplicate values for a single entity in the class, it should be declared as **DISTINCT**. Note that the **UNIQUE** option prevents duplicates among values for the entire class; hence it is stronger than the **DISTINCT** option. For a multivalued EVA, the **DISTINCT** option is useful only when the inverse EVA is also multivalued. In this many-to-many case, **DISTINCT** on either EVA implies **DISTINCT** on both EVAs and prevents duplicate relationship instances. Since duplicate instances within a many-to-many relationship are rarely useful, all EVAs participating in many-to-many relationships should be declared as **DISTINCT**.

- **MAX (MV attributes only)**

If there is an upper limit to the number of values that an entity can have for a multivalued attribute, specify this limit using the **MAX** option.

- **INVERSE (EVAs only)**

Specify the **INVERSE** option for each EVA in the schema. In addition to the advantages discussed in Section 5, “Classes,” declaring **INVERSE** options may help you find redundant relationships.

- **Subranges (DVAs only)**

Many DVAs (and almost all arithmetic DVAs) have an allowed set of values that can be used. For example, project-no probably cannot be negative and probably cannot exceed some number of digits in length. Therefore, it should be declared with a subrange such as (1 .. 99999). Subranges should be used generously throughout the schema.

As you become more familiar with attribute options and SIM database design, you probably will be able to specify most attribute options in step 1 of the logical design phase.

### User-Defined Types

As described in Section 4, “Types,” you can use user-defined types to increase the readability of the schema and to make schema changes easier. Additionally, user-defined types are essential for sharing a constructor type (**SYMBOLIC** or compound) among multiple attributes. For these reasons, you should use user-defined types whenever it is practical to do so.

Some observations and suggestions on declaring user-defined types are as follows:

- User-defined type names must be unique from all other names used in the database. This rule is explained in Section 3, “Basic ODL Considerations.”
- As with other declaration types, be sure to use the official comment to document the purpose or use of a user-defined type.
- If you name classes singularly, name user-defined types plurally or include the suffix *-type* in the name. This practice makes user-defined type names easily distinguishable from class names.

- Attributes that hold the same type of data are probably redundant and should be combined as discussed under “Step 2: Generalize and Specialize.” User-defined types should not be used to eliminate this kind of redundancy. However, user-defined types are effective in defining common data formats for such things as money and identifiers.
- User-defined types are useful for building up hierarchies of SYMBOLIC data values, such as days of the week, colors, or status codes.

Example user-defined types that employ these suggestions are shown in the ORGANIZATION database schema presented in Section 3, “Basic ODL Considerations.”

### UNIQUE Indexes

In general, indexes are used for performance reasons, and you should therefore ignore them until the physical design phase. However, there is one case in which you can use an index to add an integrity constraint to the schema. The UNIQUE attribute option prevents duplicate values for a single attribute. If a set of two or more attributes must not have duplicate values when the set is considered as a whole, you can use a UNIQUE index to enforce this constraint.

For example, suppose you have a class named Schedule that contains the DVAs schedule-date and schedule-time. Perhaps schedule dates and schedule times can be duplicated among entities. Suppose, however, that no schedule date/time combination can be duplicated. This constraint could be enforced with a UNIQUE index as shown below:

```
CLASS Schedule
  (schedule-date      : DATE, REQUIRED;
   schedule-time      : TIME, REQUIRED;
  );

INDEX Schedule-index ON Schedule
  (schedule-date, schedule-time) UNIQUE;
```

Note that, as explained in Section 6, “Indexes,” secondary keys of a UNIQUE index (schedule-time in the above example) must be REQUIRED.

### Verifies

The verify is the most general and powerful form of integrity constraint offered by SIM. A wide range of business and application requirements can be stated as verifies and subsequently enforced by SIM. In the database design process, you should use verifies extensively to capture and document rules about the persistent data within the database. When performance is critical, some verifies might need to be “turned off” as described in Section 7, “Verifies,” but performance should be ignored during logical database design.

Some observations and suggestions on the use of verifies are as follows:

- Verifies must have unique names, as described in Section 3, “Basic ODL Considerations.” One approach is to attach the prefix *Valid-* to each verify name to indicate its purpose and prevent conflicts with other identifiers used in the schema.
- Although the ELSE clause and the corresponding error message are optional for verifies, you should always use these options to produce a good error message when the verify is violated. Also, the error message helps to document the function of the verify, thereby reducing the need for an official comment.
- In general, you should not use verifies to express role-related behavior. Hierarchies automatically imply constraints relative to the roles within the hierarchy.
- Some attribute options, such as REQUIRED, MAX, and subranges, can also be expressed as verifies. In general, it is better to express these constraints as attribute options because they communicate specialized semantics that SIM can exploit when mapping the logical schema to a physical schema. However, one advantage of expressing these constraints as verifies is that you can thereby control the error message that is generated when the constraint is violated.
- UNIQUE and DISTINCT can be expressed only as attribute options; therefore, verifies are not an alternative to these constraints.
- Watch out for contradictory constraints caused by verifies. For example, consider the following schema:

```
CLASS Employee
  (employee-roles   : SUBROLE (Project-Employee, Manager), MV;
   salary           : NUMBER [8, 2];
  );
...
VERIFY Valid-Project-Employee-salary ON Project-Employee:
  salary < 50000;
...
VERIFY Valid-Manager-salary ON Manager:
  salary > 50000;
```

In this example, the subrole DVA employee-roles allows an Employee entity to be both a Project-Employee and a Manager. In actuality, however, this could never happen because an employee would have to satisfy both verifies, whose expressions are mutually exclusive.

A number of example verifies are shown in the ORGANIZATION database schema presented in Section 3, “Basic ODL Considerations.”

### Security

As discussed in Section 8, “Security,” there are two declarations that you use to define security requirements in a SIM database: accesses and permissions. By default, all information in the database is public to any program that can successfully open the database. However, once one or more accesses or permissions are declared, security is automatically enforced as specified in the security declarations.

Database security is a broad topic that cannot be adequately addressed here. However, the following guidelines might be useful when you consider the security requirements of your database:

- Although security is considered part of the logical database design, you may want to delay security specifications until one of your final iterations through the schema. This delay minimizes the number of times that you must change security specifications while fine-tuning the logical database design.
- Each access and permission must have a unique name, and each can benefit from the use of an official comment.
- Accesses and permissions are enforced automatically without surfacing in the OML. Consequently, SIM security can be changed without affecting users or application programs.
- Accesses are generic in nature, which allows you to design accesses that reflect user “types” without knowing precisely which users will fulfill the role of a particular user type. For example, you could define the following:
  - A security administrator access in which the entire database can be inquired upon
  - A project manager access in which all projects can be inquired upon or updated
  - An employee access in which only employee numbers can be accessed
  - A payroll administrator access in which employee information including salary can be retrieved or updated

The task of attaching specific accesses to individual users or programs can be performed later, possibly by a security administrator who has the responsibility for performing this task.

A number of example access and permission declarations are shown in the ORGANIZATION database schema presented in Section 3, “Basic ODL Considerations.”

### Step 4: Evaluate and Refine

After performing logical design steps 1 through 3 for the first time, you have your first version of the SIM database schema. In step 4, you evaluate the completeness and accuracy of your schema and make adjustments where necessary. The goal of this step is to fine-tune your schema to ensure that it adequately meets the persistent data requirements of your application.

#### Matching Requirements to the Schema

There are several ways in which you can evaluate your schema in terms of application requirements. If the requirements of your application system are formally documented, one approach you can take is to trace each persistent data requirement to the schema constructs that realize it. Any requirement that is incompletely or inaccurately represented should result in an adjustment to the schema.

Alternatively, you can translate each schema construct into a corresponding set of requirements that the appropriate users can then verify or adjust as necessary. For example, you could derive the following requirements for the Project class:

- All projects must have a project number.
- Project numbers must be between 1 and 99999, inclusive.
- Each current project must have a project team.
- A project team must have between 2 and 20 members.

Whichever approach you take in evaluating your schema, keep in mind the value of prototyping as a means to verify the important portions of your database design as early as possible.

Some additional suggestions for evaluating and refining a SIM database schema are described in the following paragraphs.

#### EVA/Attribute Ratios

In most applications, all data in a database are heavily interrelated. Consequently, in a properly designed SIM database, you should use many EVAs, representing relationships between classes. Precise ratios depend on your particular application requirements. However, if your EVA/attribute ratio is low, suspect redundant attributes, improper merging of different entity types into a single class, relationships not expressed as EVAs, or other problems. All relationships should be expressed through EVAs and not through other techniques that would deprive the database of semantic information.



**Information = Data + Rules**

When comparing two schema alternatives, compare the amount of information that can be deduced from each approach. Information can be characterized as the number of facts reflected in the schema, where a fact is a data element plus a rule applied to that data element. For example, consider the following schema:

```
CLASS Person
  (name           : STRING [20];
   children       : STRING [20], MV;
   employee-id    : INTEGER;
   employee-manager : Person;
  );
```

From this schema, you can deduce the following facts:

- The database is to store person entities.
- Some persons have a name, which is a string of up to 20 characters.
- Some persons have one or more children, each of which is a string of up to 20 characters.
- Some persons have an employee-id, which is an integer.
- Some persons have an employee-manager, who is a person.

Now consider the following schema:

```
CLASS Person
  (name           : STRING [20], REQUIRED;
   children       : Person, MV (DISTINCT),
                  INVERSE IS parents;
   parents        : Person, MV (DISTINCT, MAX 2);
   employee-status : SUBROLE (Employee, Previous-Employee);
  );

SUBCLASS Employee OF Person
  (employee-id    : INTEGER (1 .. 99999), REQUIRED;
   employee-manager : Employee;
  );

SUBCLASS Previous-Employee OF Person;
```

Technically, this schema stores exactly the same types of entities with the same attributes as the previous schema. However, consider the number of facts derivable from this schema:

- The database is to store person entities.
- Persons can also be employees or previous employees.
- A person does not have to be an employee or a previous employee.
- A person can be an employee or a previous employee, but not both.

- All persons must have a name, which is a string of up to 20 characters.
- Some persons have one or more children, each of which is a person.
- The inverse of children is parents, meaning that if a person has children, the children have parents.
- The children/parents relationship is a many-to-many relationship.
- Parents cannot have the same children relationship twice.
- A child cannot have more than two parents.
- When a person is an employee, he or she must have an employee-id, which is an integer.
- Employee-id must be between 1 and 99999, inclusive.
- When a person is an employee, he or she can optionally have an employee-manager, who must be another employee.
- When a person is a previous employee, no information other than person is stored.

Although the two schemas store the same amount of data, the second schema has many more rules, thereby increasing the amount of information known about the data. The second schema results in a far more accurate, reliable, complete, and usable database.

### Unnecessary Classes

You may occasionally find a class that has too little information to warrant a separate class. For example, suppose that you defined the project team of the ORGANIZATION database as follows:

```
CLASS Project          "Current projects in progress"
  (current-project-team "Current project team group"
    : Project-Team,
    INVERSE IS team-project;
  );

CLASS Project-Team     "Project teams for projects"
  (team-project        : Project,
    INVERSE IS current-project-team;
  team-employees       "Current employees on project"
    : Employee, MV;
  );
```

This example is a perfectly legitimate way to model the project team. However, suppose that you cannot find any more properties of the Project-Team class, even after several iterations through the schema. Since the EVAs current-project-team and team-project form a one-to-one relationship, each Project-team entity corresponds to precisely one Project and vice versa. Consequently, you must consider that Project-Team is really a property of each project and does not warrant a new class. Therefore, you should eliminate the Project-Team class and, instead, declare a multivalued attribute of Project, such as project-members, as shown in the following syntax:

```
CLASS Project          "Current projects in progress"
  (project-members     "Current employees on project"
   : Employee, MV;
  );
```

Note that when the *information = data + rules* formula is used to compare these two project team approaches, both contain basically the same amount of information.

### Nonrequired Attributes

An attribute that is not designated as REQUIRED can be null for some entities in the class. A null attribute might indicate that a value is legitimately optional, such as children, or that it is occasionally unknowable, such as birthplace. On the other hand, it might indicate that the attribute is not applicable for some entities in the class. In this case, the attribute may actually belong to a subclass that may or may not yet be defined.

For example, suppose you recognize that current-projects of Employee does not always apply because some employees do not work on projects. You should declare a subclass of Employee that represents employees who work on projects (for example, Project-Employee). Then, you should move the current-projects attribute to this subclass. You can now assert that every Project-Employee entity must have a value for current-projects. Consequently, you can make current-projects REQUIRED.

### Fringe Conditions

Test the database for its ability to handle fringe conditions, namely, borderline conditions and exceptions. For example, you could ask such questions as the following about the Project class:

- Can a project have more than one manager?
- Can a project have no manager?
- Can a project have no project team?
- Can a nonmanager be a project manager?
- Can a project have an arbitrarily large project team?
- What happens to a project when it is completed?
- Can a project be completed but restarted?

You might find an exception condition that is actually fairly common (for example, projects that have multiple managers). In this case, you should probably change the schema to reflect the condition as a normal case. However, if a certain condition occurs very rarely, you might not want to “pollute” the schema by allowing the exception as a normal variation. Instead, you might want to create new subclasses that reflect unusual roles, or you might want to tie certain verify conditions to status code attributes.

For example, suppose that a salary can exceed the maximum allowed salary only if certain approval has been made. You could reflect this constraint with the following schema:

```
CLASS Employee                "Current employees of the company"
  CLASS-ATTRIBUTES
    (maximum-salary           "Maximum salary any employee can have"
      : NUMBER [8, 2];
    )
    (salary                   "Salary of employee"
      : NUMBER [8, 2], REQUIRED;
    salary-exception          "TRUE when special maximum salary "
                              "exception has been approved"
      : BOOLEAN, REQUIRED, INITIALVALUE FALSE;
    );

VERIFY Valid-employee-salary ON Employee:
  salary LEQ maximum-salary OR salary-exception
  ELSE "Employee salary can not exceed the maximum allowed "
      "without special approval";
```

As shown, each employee’s salary must be below the current maximum, or else he or she must have the salary-exception = TRUE. For security purposes, you could protect the salary-exception flag from being modified except by the appropriate users and/or programs.

### Many-to-Many Relationships

Many-to-many relationships rarely exist because usually each relationship instance must have some kind of relationship data associated with it. As described in “Step 1: Identify Things and Their Properties,” this condition breaks the relationship into two one-to-many relationships with an “intersection” class holding the relationship attributes. Therefore, if you find a many-to-many relationship, determine if there is some hidden relationship data that has not yet been properly defined.

If you discover a legitimate many-to-many relationship, remember that it probably should not allow duplicate relationship instances. Accordingly, you should declare one or both of the corresponding EVAs as DISTINCT.

### Isolated Islands of Data

Since all data in the database should be related to other data, an isolated class or hierarchy (which has no relationships to other classes) should rarely occur. If you find such a situation, consider that some relationships may not have been properly declared using EVAs, or the corresponding data may not belong in the database.

### Tradeoffs and Special Cases

Some suggestions on how to handle certain design alternatives and special situations are presented in the following paragraphs.

#### Multiple Inheritance

Occasionally, you may find that a subclass needs multiple superclasses. For example, an Interim-Manager subclass may be identified in which each entity must be both a Manager and a Project-Employee. These multiple-superclass subclasses are supported by SIM, and they need a few extra considerations:

- The subclass inherits all attributes of all of its superclasses. Consequently, the set of all of these names must be unique.
- The subrole DVA of the least common ancestor (LCA) for the subclass must be multivalued.
- For an entity to participate in the subclass, it must participate in all the superclasses of that subclass.

#### Compounds versus Classes and EVAs

The compound data type is useful in some situations, although it can always be replaced by an EVA pointing to an appropriate class. The latter approach is more general and is recommended. Refer to Section 4, “Types,” for more information on the use of compounds.

#### Null Attributes versus New Subclasses

Although it is recommended that you specify as many attributes as possible as REQUIRED, it is probably not useful to define a new subclass for each attribute merely to allow the attribute to be specified as REQUIRED. In fact, this technique can be viewed as merely moving the “not required” constraint to the new subrole DVA introduced. You should balance the need to reduce nonrequired attributes with the proliferation of subclasses that are not really significant to the application.

#### Transitional Constraints

All SIM constraints, including attribute options and verifies, must be satisfied at the end of each OML update query. An OML query that attempts to violate a constraint is aborted, and the corresponding update is rejected.

However, you can identify some constraints that must be met at some point in time but that might not be satisfied at the end of each OML update query. For example, perhaps a project must have at least two project team members and a Project-Employee must work on at least one project, but you cannot always satisfy both of these conditions at all times.

One way to handle this requirement is to introduce a “state” or “status” attribute that is tied to the corresponding verifies. For example, the minimum project team size of a project could be triggered when a Boolean project-active attribute of Project is true. The corresponding verify might be as follows:

```
VERIFY Valid-project-team-minimum ON Project:  
  NOT project-active OR COUNT (project-members) GEQ 2  
  ELSE "Active projects must have at least 2 project-members";
```

The maximum salary exception previously described, under “Fringe Conditions,” is another example of this technique.

### Historic Data

Very often, you may find the need to keep a past history on certain information. For example, the ORGANIZATION database must keep track of previous employees and completed projects for current employees. Some techniques for tracking historic data are summarized as follows:

- Version attributes

An attribute can be used to indicate version information for entities, such as revision number, entry date, release number, and so on. Each time a new revision of an entity is made, a new entity can be inserted with the appropriately updated version attribute. Queries can then be directed at the “latest” version (for example, highest) value or a specific version level.

- Special subclasses

Instead of deleting obsolete entities, you can add them to a special subclass that represents an historic state. Concurrently, they may be deleted from subclasses where they are no longer needed. For example, in the ORGANIZATION database, an employee that leaves the company is deleted from all subclasses but is retained in the Person class and is added to the Previous-Employee subclass.

- Separate hierarchies

Historical data can also be handled by completely deleting entities from the hierarchies that represent their “current” form and adding representative entities in a new class.

### Conflicting Views

Sometimes you might find that different users have conflicting or competing views on an application requirement. Each user might have a biased and/or incomplete understanding of the area in question. One approach to solve this situation is to create a schema that represents an intersection of all competing views and then simplify the schema by reducing redundancy, moving attributes, and performing other refinements that you normally make. The resulting process should expose any inconsistencies and force a composite schema that adequately reflects all user requirements.

## Interface Design

The purpose of the interface design phase is to focus on how the database will be used by users and programs to determine if additional schema refinements are necessary. When you consider the query and transaction processing requirements of your database, you may identify schema additions or modifications that are necessary.

The interface design phase consists of the following steps:

- Step 1: Identify ad hoc query requirements.
- Step 2: Identify programmatic query requirements.
- Step 3: Evaluate and refine.

In steps 1 and 2, you identify and specify how ad hoc query users and application programs are intended to use the database. In both steps, you define OML queries that represent expected user and program usage of the database. The purpose of step 3 is to verify these queries using any of several techniques. By formulating and verifying OML queries, you identify schema changes that are needed to accommodate expected database usage.

Before you begin the interface design phase, you should be familiar with OML so that you can define and test queries against the database.

### Step 1: Identify Ad Hoc Query Requirements

Depending on your particular application system, the ad hoc query requirements for your database can be minor or extensive. Most application systems require some ad hoc query access. Administrative business applications typically require moderate ad hoc query access. Executive business applications can require very sophisticated ad hoc query access. Your database will almost certainly require ad hoc retrieval access, and it might require ad hoc update access as well. The purpose of this step is to identify the ad hoc query requirements of your application system and to ensure that your database allows those requirements to be met.

Ad hoc queries are those that are dynamically constructed and submitted by users. Typically, such users use a query product such as IQF or WQF, or they use a tailored application program that provides ad hoc query capabilities. In either case, the user can use a combination of queries constructed “from scratch” and queries that are predeclared and saved. In addition, many ad hoc query users prefer to construct their own tailored queries and save them for later reuse.

To ensure that your database meets the needs of ad hoc query users, you should identify the types of retrievals and updates that these users expect to perform. These needs might be enumerated in the requirements documentation for your application system, or you might have to derive them as part your database development process.

Some suggestions for defining ad hoc query requirements are presented in the following paragraphs.

### Capturing Queries

The best way to capture ad hoc query requirements is to construct representative OML queries. Some of the techniques that you can use are summarized as follows:

- Queries can be simply “hand written.” This technique allows queries to be visually verified, but it makes queries harder to actually execute against the database.
- Queries can be constructed by using IQF or WQF and retained with the appropriate query save mechanism. This technique allows queries to be verified against the actual database.
- Queries can be incorporated into a text file and processed by a special-purpose program using the OML library interface. This technique allows queries to be easily viewed and printed, and it allows them to be executed against the database for verification.

An advantage of the last two techniques is that a prototype database can be used to verify the expected behavior of each OML query.

### Update Queries

Be aware that each update query submitted through IQF or WQF is processed as a separate transaction. Consequently, the results of each update query become immediately available to other users of the database. Therefore, updates that require multiple queries bracketed within a single transaction should be submitted by application programs in which the transaction state can be more finely controlled.

Also, ad hoc update queries are much more likely to attempt to place erroneous data into the database. Consequently, you must ensure that sufficient integrity and security constraints are specified to protect the database from incomplete or incorrect updates.

### Retrieval Queries

Ad hoc retrieval queries must return data in a format that is identifiable to users. For example, a string that contains multiple elements, such as name of Person, is more understandable if it is divided into separate attributes, such as first-name, last-name, and middle-initial. Also, numeric codes that represent status or similar information are more intelligible when expressed as symbolic or string data types than as arithmetic data types.

Information returned in one query is often used as input to a subsequent query. For this reason, certain entities might require sufficient identification so that they can be unambiguously referenced in another query. For example, if person names are not REQUIRED and UNIQUE, you might want to consider maintaining an additional attribute to ensure that persons can be uniquely identified.



## Step 2: Identify Programmatic Query Requirements

Most databases are complemented by application programs that perform specialized processing. Most application systems use programs to perform the majority of database updates, even if ad hoc update queries are allowed. In addition, tailored application programs are often used to generate specialized reports. The purpose of this step is to ensure that your database sufficiently addresses the processing requirements of your system's application programs.

As with ad hoc query users, to ensure that your database sufficiently meets the needs of application programs, you should identify the types of retrievals and updates that these programs are expected to perform. These needs might already be formally documented, or you might have to derive them as part your database development process.

Some suggestions for defining programmatic query requirements are presented in the following paragraphs.

### Capturing Queries

As with ad hoc queries, the best way to capture programmatic query requirements is to use OML queries. The same techniques that can be used for capturing ad hoc queries are also applicable to programmatic queries. Also, scaled-down prototype programs that simulate key query processing scenarios can be very useful for defining and testing program requirements.

### Update Queries

Application programs typically perform a series of update queries in a single transaction. Consequently, you should define both individual queries and complete transactions that represent the types of updates that application programs are expected to perform.

Since application programs typically perform more complex updates than do ad hoc query users, they might require additional persistent data to assist their processing. For example, suppose that each new project must have a project number that is a unique, increasing, sequential integer. In order to keep track of the next project number to be used, you might want to add a class attribute to the Project class. This attribute would be incremented by each transaction that inserts a new Project entity. The corresponding ODL syntax is the following:

```
CLASS Project          "Current projects in progress"
  CLASS-ATTRIBUTES
    (next-project-no   "project-no value for next Project"
      : INTEGER (1.. 99999);
    )
    (project-no        "Uniquely identifies each project"
      : INTEGER (1.. 99999), REQUIRED, UNIQUE;
    );
```

Other examples of additional persistent data that might assist application programs are the following:

- Client information, such as the current users, stations, or sessions that are being serviced
- Restart information, such as the last input record processed
- Summary information or other results that will be retrieved by ad hoc users or other application programs

### Data Types and Formats

Certain programming languages might not be able to accommodate all of the data types that SIM supports. For example, some languages might not support NUMBER data types, which are expressed in fixed-point, 4-bit BCD format, or KANJI STRING data types, which contain 16-bit characters. Be sure that all attributes that are to be updated and/or retrieved by an application program have data types that are usable by the corresponding programming language, or that some technique will be used to map SIM data formats into formats usable by the program. Note that some SIM data types (such as DATE, TIME, SYMBOLIC, and SUBROLE) can be expressed in a numeric format or a string format for a given query, depending on the options used when the query is compiled.

Another consideration you should be aware of is precision. Application programs might be required to process data with different arithmetic precisions than are required in the database. For example, a program might process all dollar amounts by using a NUMBER [10, 2] format, whereas the database is required to store dollar amounts in only a NUMBER [8, 2] format. STRING data type sizes could also differ between application programs and the database. These differences might not be significant, depending on how data is moved between the database and the application program. However, in some cases you might want to consider adjusting the precision of certain attributes stored in the database to match the precision used by application programs.

### Step 3: Evaluate and Refine

Once you have identified ad hoc and programmatic query requirements, you should verify the corresponding OML queries. Optimally, you should use a prototype database, populated with realistic test data, on which you can execute each OML query and transaction you have defined. Alternatively (or additionally), you can use the text of each OML query as documentation that can be reviewed and verified by appropriate users, programmers, and other personnel. The use of prototyping is highly recommended as a means of verifying interface design criteria.

## Physical Design

The logical and interface design phases focus entirely on the logical schema requirements of your database. The physical design phase focuses on physical requirements, such as capacity, performance, and maintenance, to ensure that your database meets these requirements. Addressing physical design aspects of your database means that you add options as described in Section 1, “Introduction to ODL.”

It is important that you achieve a logically correct schema before you attempt to address physical concerns because yielding to the temptation to distort the logical schema for physical reasons would degrade the accuracy of your database. It is much easier to improve the performance of a database with a logically correct schema than to improve a logically poor schema of a high-performance database. Furthermore, SIM separates logical and physical schema constructs so that you can add or modify physical database characteristics without affecting the logical database design.

SIM provides default values for all physical database options that are usually suitable for initial development and prototyping. However, most application systems require at least some tuning of the physical options for the database in order to meet specific criteria. For example, since each SIM database is mapped to a DMSII database, most disk and memory parameters assume DMSII default values.

The physical design phase consists of the following steps:

- Step 1: Identify capacity requirements.
- Step 2: Identify performance requirements.
- Step 3: Identify operational requirements.
- Step 4: Evaluate and refine.

Before you begin the physical design phase, you should be familiar with indexes, DMSII options, and dictionary options, which are described in Sections 6, 9, and 10, respectively. These sections provide the details that you need to address physical design considerations. The following descriptions of physical design steps are primarily an overview of the topics that you should examine.

### Step 1: Identify Capacity Requirements

There are two aspects of capacity requirements that you should consider. First, you should identify the number of entities that each class must be capable of storing. Second, you should identify disk space limitations that must be observed by the database as a whole. Both of these aspects of capacity requirements are controlled by DMSII options declarations in your schema. Section 9, “DMSII Mapping Options,” provides details on how to specify the POPULATION option and other options to control the capacity constraints of your database.

Consider the following points when you adjust capacity-related options:

- Each base class is mapped to a separate disjoint data set. Since each entity in a hierarchy must participate in the base class, the POPULATION parameter for the data set for the base class determines the number of entities that the hierarchy can possess. Each subclass in the hierarchy contains the same number of entities or fewer than each of its superclasses.
- When a multivalued DVA is mapped to a disjoint data set, the POPULATION parameter for the data set controls the total number of values allowed for all entities in the class.
- The data types DATE, TIME, and SYMBOLIC require comparatively less disk space than the STRING data type. When subranges are used with the INTEGER data type, the corresponding disk space usage is less than it is for comparable NUMBER or REAL data types.
- Compaction techniques are available so that the amount of disk space used for certain types of data can be reduced.

### Step 2: Identify Performance Requirements

Your application might have such database performance requirements as the following:

- Application programs might be required to perform a certain amount of work in a given time period (for example, a certain number of transactions of a specific type per hour).
- Application programs might be required to respond with the results of a query within a certain length of time (for example, a certain search must respond within a given number of seconds).
- The database might be required to recover after a system failure within a certain length of time.

The database is not completely responsible for each of the performance criteria listed above, but it has a major impact on them. Each of these performance criteria can be affected by index declarations and DMSII options specified in your schema. Sections 6 and 9 provide details on these constructs. Some general observations and suggestions for controlling database performance are as follows:

- Memory usage has a tremendous effect on database performance. In general, you want the database to use as much memory as it needs, although this consideration must be balanced with the amount of memory available and the amount of memory required for other processes.
- Overall database throughput is also affected by several global DMSII options, such as PARAMETERS, OPTIONS, and AUDIT TRAIL ATTRIBUTES.
- Physical file placement should be considered in order to maximize the potential for performing parallel I/Os. For example, data sets, sets, and audit trails should be distributed among multiple packs instead of being grouped together on a single pack. The number of paths available to each pack, the type of media used, and other factors also influence the performance of each database file.

- The MAPPING option can be used with most SIM constructs together with other options to alter performance characteristics. The SURROGATE option can also be useful for improving the performance of classes.
- Data set and set options are available to use with many SIM constructs, such as classes and indexes. These options can be used to change structure types, block sizes, table sizes, and other criteria to improve database performance in many cases.
- A retrieval query (including the retrieve portion of an update query) is performed as a linear search unless an index is available that can be exploited by the query. Consequently, indexes can dramatically improve the performance of retrieve operations. Note the following observations and suggestions about indexes:
  - An index is automatically generated for all EVAs, providing efficient EVA traversal performance.
  - An index is automatically generated for UNIQUE DVAs, thereby providing efficient performance for equality and partial equality searching.
  - Extra indexes can and should be added on DVAs that are not UNIQUE yet are frequently used in selection expressions. In particular, identification attributes, such as name or ID number, are often used in selection expressions.
  - The mapping for any user-declared index can be controlled for greater efficiency. For example, alternative structure types, table sizes, and/or other options can often be used to improve index performance.
- The CONTROLPOINT specification for the database is the primary influence on database recovery time after a halt/load or other system failure.

### Step 3: Identify Operational Requirements

In addition to performance and capacity requirements, you should consider certain operational aspects of your database. These considerations involve the manner in which the database is to be maintained and the schema is to be managed. Some topics for consideration are presented in the following paragraphs.

#### Backup and Recovery

Every database should be periodically backed up so that it can be recovered in case of a failure. Similarly, audit trails generated by the database must be saved in case they are needed for later reprocessing. In both cases, backups should preferably be made to tapes that are stored physically away from the computer system to reduce the chance of losing all copies of the database in a single disaster.

Some observations and suggestions relative to these goals are the following:

- An AUDIT TRAIL ATTRIBUTES specification within the global DMSII options for the database can be used to specify how often audit-trail files are closed and how they are copied to tape.
- An AUDIT TRAIL ATTRIBUTES specification can also be used to cause the database to maintain two copies of the current audit trail. Alternatively, mirrored disk facilities can be used to maintain a duplicate copy of the audit trail.
- A procedure should be established to back up the database on a periodic basis and whenever a special need arises (for example, just before and after a schema change). In addition to the structure, control, and audit files of the database, the current DESCRIPTION file and DMSUPPORT library should also be backed up.

### Schema Control

As discussed in Section 1, “Introduction to ODL,” there are several advantages in making your SIM database active with respect to ADDS. You can do so by creating your database schema directly with ADDS or by specifying appropriate dictionary options through the direct ODL approach or SDA. Whether you choose to make your database active with ADDS or not, you should provide some kind of scheme in which the schema is maintained in a secure manner.

### Schema Evolution

Undoubtedly, your SIM database will require periodic changes throughout its life cycle. New requirements will need to be accommodated by the database as your application system evolves. Consequently, you should consider the long-term schema change requirements of your database. Some items to consider are the following:

- In cases where a logical schema requirement can be fulfilled in multiple ways, you might want to consider using the alternative that offers the greatest flexibility for evolution. For example, an EVA pointing to a class is more flexible than a compound DVA. Similarly, a multivalued EVA pointing to a class is more flexible than a multivalued DVA.
- Similarly, some physical mappings affect the flexibility with which the corresponding logical constructs can be modified. For example, a multivalued DVA mapped to a disjoint data set is more flexible than one that is mapped to an embedded data set or an occurring item.
- Section 12, “Changing a SIM Database Schema,” lists the restrictions and ramifications of making SIM schema changes. It also discusses how many change restrictions can be circumvented by performing a series of schema changes and OML updates.

## Step 4: Evaluate and Refine

As with any change you make to your schema, you should verify the adequacy of any physical schema change you make. If you modify the schema to accommodate a capacity, performance, or operational requirement, you should verify that the change accomplished what it was intended to do. Criteria such as performance or disk space usage often cannot be completely verified until at least a prototype test is performed.

The following suggestions for evaluating physical design changes are offered:

- If you use a prototype to verify the design of your database, be sure to verify physical requirements as well. For example, you should populate key classes to their intended capacities, including average populations for subclasses and multivalued attributes. Similarly, you should also verify the performance of critical update or retrieval queries during prototyping so that problems are discovered early in the development cycle. In particular, the OML queries that you developed in the interface design phase should be used to test database performance.
- Virtually all of the DMSII monitoring and analysis tools are also applicable to SIM databases. For example, you can use STATISTICS and DMMONITOR to analyze database usage characteristics, and you can use DBANALYZER to analyze disk space usage characteristics.
- Since all SIM databases are DMSII databases, database usage patterns might indicate the occasional need for a “garbage collection” to reclaim unused disk space. DBANALYZER can be used to determine if and when garbage collection is necessary, and the BUILDREORG facility can be used to perform garbage collections in the same manner as for DMSII databases.
- The Operations Control Manager (OCM) utility provides a simplified, menu-oriented interface to virtually all DMSII support tools, and it can be used for SIM databases as well as for DMSII databases. OCM can help you to successfully use DMSII support tools without requiring detailed knowledge of input syntax and program parameters.

# The SIM Data Model beyond Databases

The SIM data model is, of course, very proficient for designing databases. This data model can be used for designing other application components as well, owing to such rich modeling features as generalization hierarchies, complex relationships, and integrity constraints. Such features allow the SIM data model to express problem and solution elements in many functional areas, even if those elements do not correspond to persistent data as they do in a SIM database.

For example, the SIM data model can be used to represent nonpersistent data held in application programs. Some example program elements that can be modeled using ODL are the following:

- Data structures such as tables, queues, lists, and stacks  
You can use classes to represent the overall data structure and the types of entries in the data structure. Attributes can indicate fields or variables within each entry (DVAs) or references to other entries (EVAs). You can use integrity constraints to express the semantics of classes and attributes within the data structure.
- Interfaces, including entry points, parameters, and results  
You can use a class to represent an entry point. Its attributes can indicate the parameters given to the entry point and the results returned by the entry point. Complex parameters or results can appear as EVAs to new classes that represent the structure of the parameter or result.
- User-interfaces, such as windows  
You can use a class to represent a display, data-entry, pop-up, or other type of window. Attributes of the class can represent window characteristics such as position, color, and contents. EVAs can represent relationships to other windows or other data structures within the program.
- Reports  
You can use a class to represent either a printed report or a displayed report. Attributes can represent report contents, layout, and relationships to other data structures.

There are many advantages of using a common data model for as many components as possible when developing a complete application system. Besides reducing the number of models that developers must learn and communicate with, a common data model also increases the integration of individual application components. When the SIM data model is used as a documentation standard in both requirements and design documents, communication between each development phase is improved, with less loss of semantics than would result, for example, with structured development techniques.

Another advantage of using the SIM data model as a common data representation model throughout your application is that you can take advantage of various InfoExec tools. For example, the SIM Design Assistant (SDA) can be used to develop a schema even if the schema does not correspond to a SIM database. ODL syntax and semantic checks can still be verified, and the SDA graphical browsing and reporting capabilities can be used. The schema can even be generated as a SIM database and tested with OML queries to help solidify a design before application code is actually written.



Another facet to consider is that you can artificially extend the SIM data model to reflect new constructs while still adhering to standard ODL syntax. For example, object-oriented operations can be reflected in the schema by representing parameters in official comments, such as the following:

```
CLASS Employee
  (name          "DVA: First and last name of employee"
    : STRING [20];
    birthdate    "DVA: Birth date for pension purposes"
    : DATE;
    age          "Operation: Derived by subtracting "
    "current-date from birthdate"
    : INTEGER;
    give-raise   "Operation (amount: REAL): returns TRUE "
    "if raise was allowed"
    : BOOLEAN;
  );
```

In this example, the Employee class has two stored attributes (name and birthdate), a derived attribute (age, which is a form of operation), and a parametric operation (give-raise).

By using this “extended” approach, you can use the SIM data model as the basis for a complete object-oriented development methodology. Object-oriented analysis and design can be used for all application components, including the SIM database.



# Section 3

## Basic ODL Considerations

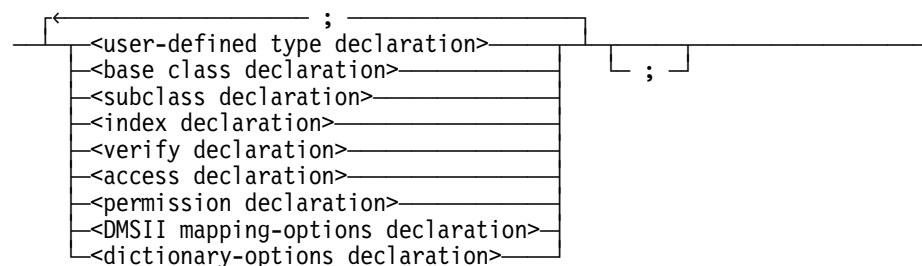
This section presents information that you should be aware of before you consult subsequent sections and begin using ODL to define SIM database schemas. The information is presented as follows:

- The kinds of ODL declarations that you can include in a schema file are described.
- The means and media that you can use to create, amend, or view the schema file are described.
- The syntax rules and the language conventions for using ODL are presented.
- The constant data values that are used in ODL declarations are described.
- An example SIM database called the ORGANIZATION database, which is reflected in examples throughout this guide, is described. The schema is shown in its entirety.

## Kinds of ODL Declarations

When you define a SIM database schema, you can include various kinds of ODL declarations in the schema file. The following diagram shows the syntax to use for specifying ODL declarations. The kinds of declarations that you can specify are named in the diagram.

**<declarations>**



Each kind of declaration that you can include in a schema is described briefly in the following table and is described in detail in a later section of this guide.

Note that every schema must contain at least one base class declaration. Inclusion of other declarations in a schema depends on the particular application needs for a given database.

| Kind of Declaration                 | Description  |
|-------------------------------------|--|
| <user-defined type declaration>     | Allows you to define a data type in a schema. You give this data type a name that distinguishes it from classes and from other user-defined types. For more information, refer to Section 4, "Types."                          |
| <base class declaration>            | Allows you to define a base class, including all the attributes that characterize the entities of that base class. For more information, refer to Section 5, "Classes."  |
| <subclass declaration>              | Allows you to define a subclass, including all the attributes that characterize the entities of that subclass. For more information, refer to Section 5, "Classes."  |
| <index declaration>                 | Allows you to define an index on a base class or a subclass, namely, to identify data-valued attributes (DVAs) that are to serve as keys for data retrievals. For more information, refer to Section 6, "Indexes."             |
| <verify declaration>                | Allows you to establish conditions that must be satisfied by the database. For more information, refer to Section 7, "Verifies."   |
| <access declaration>                | Allows you to define security constraints on a database. For more information, refer to Section 8, "Security."   |
| <permission declaration>            | Allows you to grant the constraints that have been defined by an access declaration to particular users and programs. For more information, refer to Section 8, "Security."  |
| <DMSII mapping-options declaration> | Allows you to define DMSII mapping options to optimize the physical characteristics of a database. For more information, refer to Section 9, "DMSII Mapping Options."  |
| <dictionary-options declaration>    | Allows a schema that is generated through the SIM Utility or the SIM Design Assistant (SDA) to be integrated with the Advanced Data Dictionary System (ADDS). For more information, refer to Section 10, "Dictionary Options." |

## Creating and Managing a Schema File

You can create, amend, or view a database schema file at your terminal by means of the Command and Edit language (CANDE) or the Editor program. You create the file as a text file (for example, SEQDATA or TEXTDATA), following the syntax rules and language conventions presented later in this section. For information on the use of CANDE, refer to the *CANDE Operations Reference Manual*. For information on the use of Editor, refer to the *Editor Operations Guide*.

When you have finished work on the schema file, you can then use the appropriate SIM Utility command to process the schema and put it into effect, namely, to create or update the database. To create a new database, use the ADD command. To change the schema of an existing database, use the CHANGE command. For information on use of the SIM Utility, refer to Section 11, “SIM Utility.”

## Syntax Rules

The following syntax rules apply to entering ODL syntax in a schema file:

- You must enter all the syntax to be processed by the SIM Utility within columns 1 through 72 of your schema file.
- You must end each declaration with a semicolon (;). You must also separate elements within some of the declarations (such as attributes within base class or subclass declarations) with semicolons, as shown in the syntax diagrams in this guide. Use of a trailing semicolon after the last declaration or the last element is optional.
- Every syntax element must be separated by one or more blank spaces, with the exception of punctuation symbols. An implicit blank is assumed between column 72 of a line and column 1 of the next line.
- You can enter syntax in upper case, lower case, or mixed case letters, except for string constants, which must have the precise casing desired.
- A SIM identifier can be any combination of letters, digits, hyphens, or underscores, except that it must start with a letter, end with a letter or digit, and be no longer than 30 characters. A letter is any of the characters A through Z or a through z as defined in the ASERIESEBCDIC(EBCDIC) coded character set. Letter case is ignored, and no distinction is made between hyphens and underscores. For example, A-B1 is considered identical to A\_b1.

# Language Conventions

The following paragraphs describe ODL language conventions, including the parts of a declaration, rules for using SIM identifiers, and the kinds of comments you can include in a schema.

## The Parts of a Declaration

All ODL declarations except for DMSII mapping-options declarations and dictionary-options declarations use the following form:

<construct type> <identifier> <comment> <definition>

Each part of a declaration is described in the following table:

| Declaration Part | Description   |
|------------------|---|
| <construct type> | The type of SIM construct that you are declaring, such as TYPE, CLASS, or INDEX.  |
| <identifier>     | A SIM identifier that uniquely names the construct that you are declaring.  |
| <comment>        | Optional remarks about the construct, such as a brief description of its purpose. This “official comment” is to be included in the schema file and saved in the SIM database directory. |
| <definition>     | The remainder of the declaration, whose syntax depends on the construct type.   |

## Rules for Using SIM Identifiers

When you specify a SIM identifier as part of a declaration, you must observe certain uniqueness rules, as described in the following paragraphs.

All SIM identifiers used for constructs of the following types must be unique within each SIM database. That is, when a SIM identifier is used for a construct of any of these types, that identifier cannot be used for any other construct in the same database:

- User-defined types
- Base classes
- Subclasses
- Entity-valued attributes (EVAs)
- Symbolic values
- Indexes
- Verifies
- Accesses
- Permissions

The following rules apply to SIM identifiers used for class attributes, data-valued attributes (DVAs), and compound components:

- Any identifier used for a class attribute or DVA cannot be used for another class attribute or DVA within the same class or subclass.
- An identifier cannot be used for two or more compound components that have the same owner and that are at the same level. For further explanation of this rule, refer to “Compound” in Section 4, “Types.”
- A subclass cannot inherit a DVA whose identifier is the same as that of one of the attributes defined for that subclass. Moreover, a subclass cannot inherit two or more DVAs with the same identifier. For further explanation of these rules, refer to “Generalization Hierarchies” in Section 5, “Classes.”

### Kinds of Comments You Can Use in a Schema

You can use two kinds of comments in a schema as documentation aids:

- Official comments
- Escape comments

These kinds of comments are described in the following paragraphs.

#### Official Comments

An official comment, which is referred to simply as <comment> in the ODL syntax throughout this guide, is always allowed immediately after the SIM identifier used in a declaration. You must enclose this kind of comment in quotation marks (" "). If the comment extends to more than one line, you must enclose the portion of the comment on each line in quotation marks. The total length of the comment (the total of the characters between quotation marks on all lines of the comment) cannot exceed 255 characters.

The following example shows an official comment:

```
CLASS Person    "Person has the following subclasses:"  
                "Previous-Employee and Employee."
```

An official comment is saved in the directory of the SIM database. The comment is shown in the output produced through the SIM Utility LIST command. Moreover, it can be displayed through query products such as the Interactive Query Facility (IQF) and the Workstation Query Facility (WQF). Therefore, you can use an official comment to provide a brief explanation of a declaration to database users as well as programmers.

#### Escape Comments

You can include an escape comment on any line of a schema. This kind of comment consists of a percent sign (%) followed by free-form text that can extend to column 72 of the line. An escape comment must follow the last syntax element on the line, and it cannot be embedded within a quoted string.

You might wish to include escape comments on successive lines of a schema to form one item of information, as shown in the following example:

```
CLASS Person    "Person has the following subclasses:"  
                "Previous-Employee and Employee."  
    (name        : Name-of-person, % (first name, middle initial,  
                REQUIRED          % and last name ûin that order)
```

When the SIM Utility encounters a percent sign in a schema, the remainder of the input line is ignored. Therefore, escape comments are not stored in the database directory. They only serve as documentation to someone who reads the schema file directly.



## Constant Data Values

Some declarations require the use of constant data values, such as integers and strings. For example, you might need such values to specify initial values and data type subranges in a schema. The constant data values defined for ODL are the following:

- Integer
- Real
- Character
- Kanji
- Boolean
- Time
- Date
- Double
- String

The constant data values are described in the following paragraphs. Note that constant data values are indicated in the ODL syntax diagrams in this guide, enclosed in angle brackets (for example, <integer> and <string>). For information on the kinds of data types defined for ODL, refer to Section 4, “Types.”

### Integer

An integer is a single-precision constant value (of the data type INTEGER) consisting of an optional plus (+) or minus (–) sign followed by 1 to 12 digits. An integer cannot include a decimal point (.) or an exponent.

An integer must be between the values –549755813887 and 549755813887 inclusive.

The following are examples of integers:

0  
-12  
549755813887

### Real

A real is a single-precision constant floating-point value (of the data type REAL) that is defined in the following format:

`<mantissa>E<exponent>`

The mantissa is required. The E and the exponent are optional. The `<mantissa>` construct consists of an optional plus (+) or minus (–) sign followed by 1 to 12 digits, with an optional decimal point (.) appearing between any 2 digits. An E is allowed only when an exponent is supplied. An `<exponent>` construct consists of an optional sign followed by 1 or 2 digits.

A real must be between the values `–4.31359146674E68` and `4.31359146674E68` inclusive.

The following are examples of reals:

```
0
-0.1
+6.234E-28
4.31359146674E68
```

### Character

A character is a single 8-bit constant value (of the data type CHAR) specified with either of two formats:

- Graphic
- Hexadecimal

The use of each of these formats to specify a character is described in the following paragraphs.

#### Graphic Character Format

You can express a character graphically using quotation marks ("), as in the following examples:

```
"A"
" "
"" ""
```

As shown, to express a quotation mark as a character, you must enclose two consecutive quotation marks within two quotation marks. That is, you must specify four consecutive quotation marks.

The meaning of a graphic character constant depends on the context in which it is used. For example, if you use a character in an INITIALVALUE option, you must consider the coded character set version (ccsversion) of the corresponding data type to determine how the graphic representation of the character is to be translated to an internal binary value. If you use a character in a subrange expression, you must consider the ccsversion of the corresponding data type to determine the collating value of the character with respect to other characters.

## Hexadecimal Character Format

You can express a character using hexadecimal format, as in the following examples:

```
4"00"  
4"40"  
4"FF"
```

When you use hexadecimal format, you must place the number 4 immediately before the first quotation mark, and you must specify two hexadecimal digits between the quotation marks. A hexadecimal digit is any of the digits 0 through 9 or the capital letters A through F. The hexadecimal format allows a nongraphic character to be represented as a character constant. When you use the hexadecimal format, the corresponding hexadecimal value specifies the internal binary representation of the character, regardless of the ccsversion of the corresponding data type.

## Kanji

A Kanji is a single 16-bit character constant (of the data type KANJI) specified with the hexadecimal format, as in the following examples:

```
4"2B00012C"  
4"2BFFFF2C"
```

As shown, the hexadecimal constant must begin with the digits 2B, which form the start-of-double octet (SOD) character, and end with the digits 2C, which form the end-of-double octet (EOD) character. Between the SOD and EOD, a 16-bit value must be expressed using four hexadecimal digits. The Kanji character is considered to be the 16-bit value between the SOD and EOD. The SOD and EOD are only used to designate the string as containing a Kanji constant data value.

## Boolean

A Boolean is a constant value (of the data type BOOLEAN), of which there are only two: TRUE and FALSE. Note that letter case is irrelevant, as shown in the following examples:

```
true  
False
```

### Time

A time is a constant value (of the data type TIME) that you can express in either of two formats:

- HH:MM:SS
- HH:MM

HH represents hours, MM represents minutes, and, when used, SS represents seconds. Each of the elements HH, MM, and SS must be a one-digit or two-digit integer, and the composite value must be a valid time. That is, hours equal 0 through 23, minutes equal 0 through 59, and seconds equal 0 through 59.

The following are examples of time constants:

```
12:00:01
23:00
1:2:3
```

### Date

A date is a constant value (of the data type DATE) that you can express in either of two formats:

- MM/DD/YY
- MM/DD/YYYY

MM represents a month, DD represents a day, and YY and YYYY represent a year. Each of the elements MM, DD, and YY must be a one-digit or two-digit integer. If the MM/DD/YY format is used, the year is considered to be YY + 1900. If the MM/DD/YYYY format is used, YYYY must be a three-digit or four-digit integer, and it is considered to be the exact year desired. Whichever format is used, the composite value must be a valid date. That is, month equals 1 through 12, day equals 1 through 31, year equals 1 through 99 or 001 through 9999, and the day must be valid for the month and year given.

The following are examples of date constants:

```
1/1/90
2/29/1992
12/31/9999
```

## Double

A double is a double-precision floating-point value. A double is expressed in the same format as a real (described previously under “Real”) except that for a double the mantissa can be 1 to 24 digits long and the exponent can be 1 to 5 digits long.

A double must be between the values `-1.94882838205028079124467E29603` and `1.94882838205028079124467E29603` inclusive.

The following are examples of doubles:

```
0.1
-1E-1
+6.2345678901234567890E+999
1.9E29603
```

## String

A string is a multiple-character constant (of the data type `STRING`) that you can specify using either of two formats:

- Graphic
- Hexadecimal

The use of each of these formats to specify a string is described in the following paragraphs.

### Graphic String Format

You can specify a string graphically as a series of characters enclosed in quotation marks, as in the following examples:

```
" "
```

```
"The man said ""Hello"""
```

```
""""
```

Two successive quotation marks within a string constant are considered a single, embedded quotation mark.

A pair of quotation marks by itself (“ ”) is allowed; this is considered an empty string constant (zero-length string).

As is the case with a character, the meaning of a graphic string constant depends on the `ccsversion` of the relevant data type. If you use a graphic string constant in a subrange expression or an `INITIALVALUE` option, you must consider the `ccsversion` of the corresponding data type to determine the following:

- How the graphic string constant is represented internally
- How the graphic string constant is collated with respect to other string constants

### Hexadecimal String Format

You can specify a string in hexadecimal format, as in the following examples:

```
4"00"  
4"C1C2C3"
```

When you use the hexadecimal format, you must place a 4 immediately before the first quotation mark, and you must specify an even number of hexadecimal digits within the quotation marks. This format allows you to specify a string constant using nongraphic characters. Note that when you use hexadecimal format, the given hexadecimal value specifies the internal binary representation of the string constant regardless of the ccsversion of the corresponding data type.

### Considerations for Using Graphic and Hexadecimal String Formats

By default, both the graphic and hexadecimal formats of a string are interpreted as representing a string of 8-bit characters. However, the hexadecimal format of a string can also be used to specify 16-bit character strings. For example, use of the hexadecimal string format for 16-bit character strings is needed to specify the INITIALVALUE construct for an attribute whose data type is STRING based on Kanji.

A Kanji string is specified as follows:

- Place the hexadecimal digits 2B (the SOD character) immediately after the first quotation mark.
- Place the hexadecimal digits 2C (the EOD character) immediately before the second quotation mark.
- Place one or more sets of four hexadecimal digits between the SOD and EOD.

The following are examples of Kanji string constants:

```
4"2B00002C"  
4"2B0001000200032C"
```

You can split a long string constant across multiple schema lines using implicit concatenation. Multiple, successive graphic constants, each of which is separated by one or more blank spaces, are implicitly concatenated into a single string constant. For example, consider the following:

```
CLASS C  
  (S: STRING [200] INITIALVALUE "This is a long INITIALVALUE "  
                                "constant that demonstrates "  
                                "implicit concatenation"  
  );
```

In this example, the DVA *S* has an INITIALVALUE constant that is split across three schema lines. The SIM Utility parses the three individual string constants as if they were a single string constant. That is, the string constant in the example is actually “This is a long INITIALVALUE constant that demonstrates implicit concatenation”.

You can also use implicit string concatenation for hexadecimal constants, as shown in the following example:

```
CLASS C
  (S: STRING [200] INITIALVALUE 4"C1C2C3C4C5C6C7C8C9C0"
                                4"D1D2D3D4D5D6D7D8D9D0"
                                4"F1F2F3F4F5F6F7F8F9F0"
  );
```

When you use implicit string concatenation, you must specify each string segment using the same format. That is, you must use either the graphic format for the entire string or the hexadecimal format for the entire string. Furthermore, if the hexadecimal format is used, each string segment must have an even number of hexadecimal digits.

# ORGANIZATION Database

An example SIM database called the ORGANIZATION database is used throughout this guide to illustrate the use of ODL.

In the following paragraphs, a general description of the ORGANIZATION database is provided, including an overview of its major components. After the general description, the database schema is shown in its entirety. Note that this ORGANIZATION database is used only in the examples presented in this guide; it differs substantially from the ORGANIZATION database described in other InfoExec documents referenced in this guide.

## General Description

Figure 3–1 shows the relationships between the classes in the ORGANIZATION database. In the figure, the following conventions are used:

- Each oval represents a class.
- Class names are indicated within each oval.
- Broken lines between classes represent superclass-subclass relationships.
- Light lines with arrows represent single-valued EVA relationships between classes.
- Heavy lines with arrows represent multivalued EVA relationships between classes.
- EVA names are indicated adjacent to the light and heavy lines.



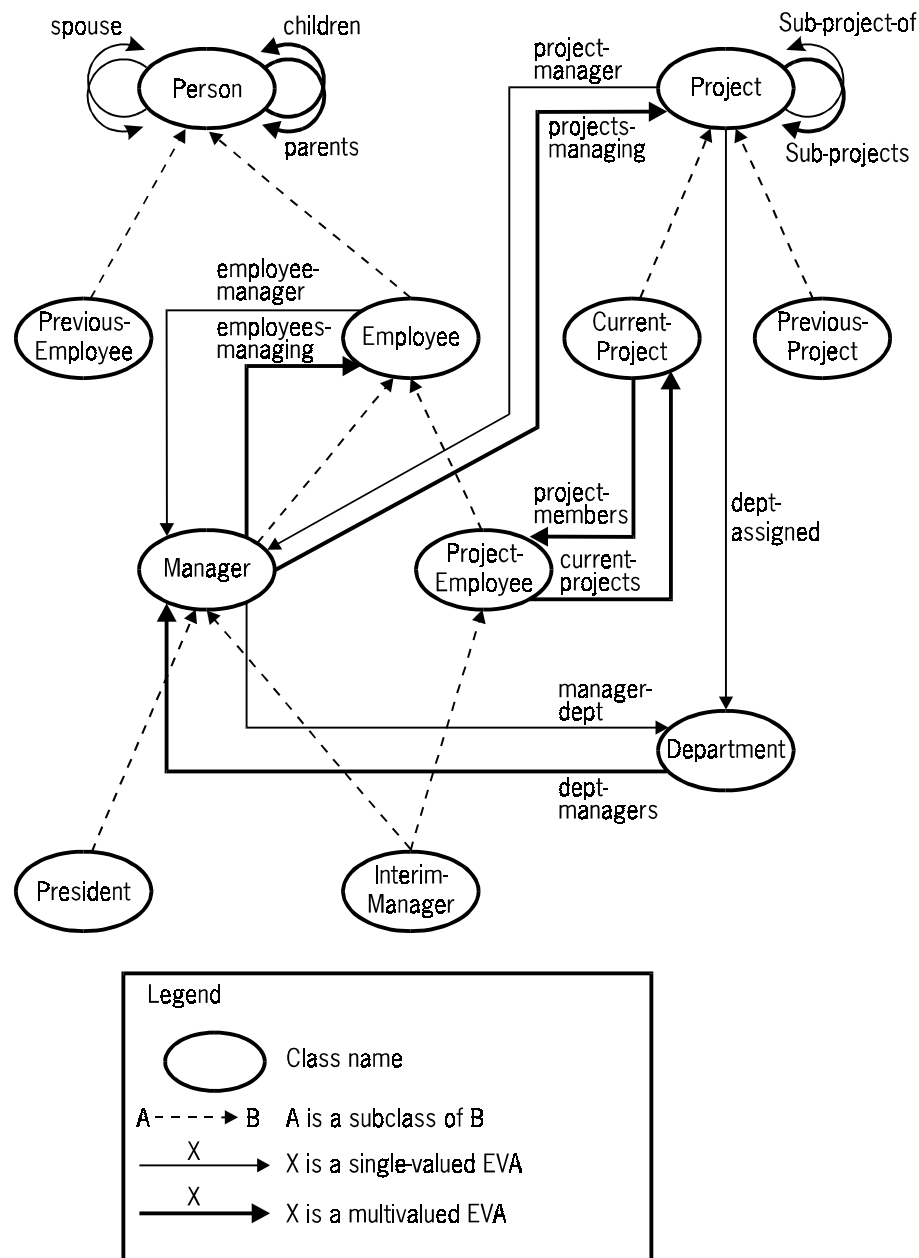


Figure 3-1. Relationships between the Classes in the ORGANIZATION Database

The ORGANIZATION database keeps track of current and previous employees and current projects within a hypothetical company. Personnel information is kept for each employee, such as their name, address, children, parents, whether they are managers, their salary, and so forth. Project information maintained includes current and completed projects, employees working on current projects, the manager of each project, subprojects that comprise a project, and so forth.

The database contains the following three base classes:

- Person
- Project
- Department

Each base class forms a separate generalization hierarchy. Therefore, each base class defines a basic type of entity that can be stored in the database.

Person has the following two subclasses:

- Employee
- Previous-Employee

Since the subrole DVA (employee-status) that defines these subclasses is single valued (the default), a Person entity can be an Employee or a Previous-Employee, but not both. Since employee-status is not designated as REQUIRED, a Person entity does not have to participate in one of the subclasses of Person.

Previous-Employee has no subclasses, but Employee has the following two subclasses:

- Manager
- Project-Employee

The subrole DVA (employee-roles) that defines these subclasses is multivalued (MV). Therefore, an Employee entity can also be a Manager, a Project-Employee, or both.

Both Manager and Project-Employee have a single subclass: Interim-Manager. Consequently, Interim-Manager is a subclass with multiple superclasses. To participate in Interim-Manager, an entity must participate in both Manager and Project-Employee.

The database has several user-defined types, such as Person-id-type and Capital-letters.

The person-id attribute of Person is a data-valued attribute (DVA). Its type is the primitive data type INTEGER.

The spouse attribute of Person is an entity-valued attribute (EVA). Since its type is the same as its owner (Person), the relationship defined by spouse is called a reflexive relationship. Since spouse is its own INVERSE (that is, both ends of the relationship are called spouse), its relationship is also called self-reflexive. Spouse forms a one-to-one relationship.

The employees-managing EVA of Manager, which is multivalued (MV), and its inverse EVA, employee-manager of Employee, which is single valued (SV), form a one-to-many relationship.

The EVAs parents and children of Person form a many-to-many relationship that is also reflexive.

The ORGANIZATION database schema contains many examples of attribute options, for example, REQUIRED, MAX, and INITIALVALUE. The schema also contains examples of the following:

- Auxiliary access methods (INDEX declarations)
- General integrity constraints (VERIFY declarations)
- Security specifications (ACCESS and PERMISSION declarations)
- Implementation options (DMSII mapping-options and dictionary-options declarations)

## Schema

The database schema shown on the following pages is the complete schema for the ORGANIZATION database. This schema is reflected in examples presented throughout this guide. Note that in a number of these examples the ODL constructs may vary (to illustrate a particular point of interest) from comparable constructs shown in the complete schema.

The casing of ODL syntax entered in a schema file other than string constants is immaterial from the processing standpoint. The following casing conventions are observed for the ORGANIZATION database in this guide simply for the sake of clarity and consistency:

- Key words are in all uppercase letters (for example, CLASS, INTEGER).
- Class names begin with an uppercase letter. If a part of a class name follows a hyphen, that part also begins with an uppercase letter (for example, Project-Employee).
- The names of user-defined types, indexes, verifies, accesses, and permissions begin with an uppercase letter. If a part of these names follows a hyphen, that part does not begin with an uppercase letter (for example, Valid-person-name).
- Attribute names, symbolic values, and component names are in all lowercase letters (for example, name, single, street).

% Schema for SIM database: ORGANIZATION

%% Person %%%%%%%%%%

TYPE Person-id-type            "Format for person-id values"  
= INTEGER (1..99999);

TYPE Capital-letters           "Uppercase letters A through Z"  
= CHAR ("A".. "I", "J".. "R", "S".. "Z");

TYPE Capitals-or-space        "Capital letters or a space"  
= CHAR (" ", "A".. "I", "J".. "R", "S".. "Z");

```

TYPE Phone-number-type      "Format for phone numbers" =
    (area-code              : NUMBER [3];
     prefix                  : NUMBER [3];
     suffix                  : NUMBER [4];
    );

CLASS Person                 "Persons related to the company"
    CLASS-ATTRIBUTES
        (next-person-id      "person-id for next new Person"
         : Person-id-type;
        )
    (employee-status         "Whether person is a current or previous "
                                "employee of the company"
         : SUBROLE (Employee, Previous-Employee);
     person-id               "Unique person identification"
         : Person-id-type, REQUIRED, UNIQUE;
     name                     "Person's first-name, middle-initial, and "
                                "last-name":
         (first-name          : STRING [15] OF Capitals-or-space;
          middle-initial       : STRING [1] OF Capital-letters;
          last-name            : STRING [20] OF Capitals-or-space;
         ), REQUIRED;
     birth-date               "Date of birth"
         : DATE (1/1/1900..12/31/1999), REQUIRED;
     marital-status           "Current marital status"
         : SYMBOLIC (single, married, divorced),
           REQUIRED, INITIALVALUE single;
     current-residence        "Current home address":
         (street              : STRING [30];
          city                 : STRING [20] OF Capitals-or-space;
          state                : STRING [2] OF Capital-letters;
          zipcode              "5 or 9 digits":
              (prefix          : INTEGER (1..99999);
               suffix           : INTEGER (1..9999);
              );
         ), REQUIRED;
     home-phone               "Home phone number (optional)"
         : Phone-number-type;

us-citizen                   "U.S. citizenship status"
                                : BOOLEAN, REQUIRED;
gender                       "Gender for tax reporting"
                                : SYMBOLIC (male, female), REQUIRED;
spouse                       "Person's spouse if married"
                                : Person, INVERSE IS spouse;
children                     "Person's children (optional)"
                                : Person, MV (DISTINCT), INVERSE IS parents;
parents                      "Person's parents (optional)"
                                : Person, MV (DISTINCT, MAX 2),
                                  INVERSE IS children;
    );

```

```

VERIFY Valid-person-id ON Person:
    person-id LSS next-person-id AND next-person-id EXISTS
    ELSE "Person-id must be less than next-person-id";

VERIFY Valid-person-name ON Person:
    first-name OF name EXISTS AND last-name OF name EXISTS
    ELSE "Person must have a first-name and a last-name";

VERIFY Valid-current-residence ON Person:
    street OF current-residence EXISTS AND
    city OF current-residence EXISTS AND
    state OF current-residence EXISTS AND
    prefix OF zipcode OF current-residence EXISTS
    ELSE "Current-residence must have a street, city, state, and "
        "zipcode prefix";

VERIFY Valid-spouse-exists ON Person:
    (marital-status NEQ married AND NOT spouse EXISTS) OR
    (marital-status EQL married AND spouse EXISTS)
    ELSE "Persons must have a spouse when they are married";

VERIFY Valid-spouse-gender ON Person:
    gender OF Person NEQ gender OF spouse
    ELSE "Person's spouse must be of the opposite gender";

VERIFY Valid-children ON Person:
    Person NEQ children
    ELSE "A person can not be his or her own child or parent";

INDEX Person-by-id ON Person
    (person-id);

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Employee %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

TYPE Employee-id-type      "Format for employee-id values"
                           = INTEGER (1..99999);

TYPE Dollar-type           "Format for dollar amounts"
                           = NUMBER [8, 2];

SUBCLASS Employee         "Current employees of the company"
  OF Person
  CLASS-ATTRIBUTES
    (next-employee-id      "Employee-id for next new Employee"
      : Employee-id-type;
      last-hire-date       "Last date an Employee was hired"
      : DATE;
      maximum-salary       "Maximum salary for employees"
      : Dollar-type;
    )

```

```
(employee-roles      "Whether Employee is a manager, project "  
                      "employee, or both"  
                      : SUBROLE (Manager, Project-Employee), MV;  
  employee-id         "Unique employee identification"  
                      : Employee-id-type, REQUIRED, UNIQUE;  
  hire-date           "Hire date for pension purposes"  
                      : DATE, REQUIRED;  
  salary              "Current yearly salary"  
                      : Dollar-type, REQUIRED;  
  salary-exception    "TRUE if salary can exceed maximum"  
                      : BOOLEAN, REQUIRED, INITIALVALUE FALSE;  
  exemption-status    "Exemption status"  
                      : SYMBOLIC (exempt, non-exempt), REQUIRED,  
                      INITIALVALUE non-exempt;  
  employee-manager    "Employee's current manager"  
                      : Manager, INVERSE IS employees-managing;  
);
```

```
VERIFY Valid-employee-id ON Employee:  
  employee-id LSS next-employee-id  
  ELSE "Employee-id must be less than next-employee-id";
```

```
VERIFY Valid-employee-hire-date    "Employees must be 18 or older"  
  ON Employee:  
  ELAPSED-DAYS (birth-date, hire-date) DIV 365.25 GEQ 18  
  ELSE "Employees must be at least 18 years old as of hire-date";
```

```
VERIFY Valid-last-hire-date ON Employee:  
  last-hire-date EXISTS AND hire-date LEQ last-hire-date  
  ELSE "Hire-date must be less than or equal to last-hire-date";
```

```
VERIFY Valid-employee-salary-1 ON Employee:  
  salary LSS salary OF employee-manager  
  ELSE "Employee's salary must be less than that of his or her "  
       "manager";
```

```
VERIFY Valid-employee-salary-2 ON Employee:  
  ABS (salary - AVG (salary OF Employee)) LEQ  
  AVG (salary OF Employee) *.2  
  ELSE "Employee salary must be within 20% of average employee "  
       "salary";
```

```
VERIFY Valid-employee-salary-3 ON Employee:  
  salary LEQ maximum-salary OR salary-exception  
  ELSE "Salary can not exceed the maximum without exception approval";
```

```
VERIFY Valid-manager ON Employee:  
  employee-manager NEQ Employee  
  ELSE "An employee can not be his own manager";
```

```

VERIFY Valid-manager-exists ON Employee:
  Employee ISA President OR employee-manager EXISTS
  ELSE "All employees (except for the President) must have a manager";

```

```

INDEX Employee-index ON Employee
  (hire-date, salary, exemption-status);

```

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Manager %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```

SUBCLASS Manager          "Managers of the company"
  OF Employee
  (manager-status         "Optional Manager roles: interim manager "
                           "or a President"
                           : SUBROLE (Interim-Manager, President);
  employees-managing      "Employees reporting to manager"
                           : Employee, MV, INVERSE IS employee-manager;
  manager-level           "Manager's management level"
                           : SYMBOLIC (supervisor, department-manager,
                                       division-manager, executive) ORDERED,
                           REQUIRED;
  bonus                   "Yearly bonus, if any"
                           : Dollar-type;
  projects-managing       "Projects responsible for"
                           : Project, MV, INVERSE IS project-manager;
  manager-dept            "Department to which manager (and all "
                           "subordinates) belong"
                           : Department, REQUIRED,
                           INVERSE IS dept-managers;
);

```

```

VERIFY Valid-managing-count ON Manager:
  (manager-level EQL supervisor AND
   COUNT (employees-managing) LEQ 10
  ) OR
  (manager-level EQL department-manager AND
   COUNT (employees-managing) LEQ 100
  ) OR
  manager-level EQL division-manager OR
  manager-level EQL executive
  ELSE "Manager manages too many people for his/her level";

```

```

VERIFY Valid-manager-bonus ON Manager:
  bonus LEQ 10000 AND bonus LEQ salary * 0.1
  ELSE "A manager's bonus can not exceed $10,000 or 10% or his or her "
      "salary";

```

%% Interim-Manager %%%

```
SUBCLASS Interim-Manager    "Employees temporarily acting as a project "
                             "employee and a manager"
    OF Manager AND Project-Employee;
```

%% President %%%

```
SUBCLASS President          "Current president of the company"
    OF Manager;
```

```
VERIFY Valid-president-count ON President:
    COUNT (President) LEQ 1
    ELSE "Only one employee can be the President";
```

%% Project-Employee %%%

```
SUBCLASS Project-Employee   "Employees who are project team members"
    OF Employee
    (project-employee-status "Whether or not Employee is acting as an "
                             "interim manager"
                             : SUBROLE (Interim-Manager);
    current-projects         "Current projects of employee"
                             : Current-Project, REQUIRED, MV (DISTINCT,
                             MAX 6), INVERSE IS project-members;
    );
```

%% Previous-Employee %%%

```
SUBCLASS Previous-Employee  "Past employees of the company"
    OF Person
    (leave-status            "Status of employee's leave"
                             : SYMBOLIC (disability, leave, retired,
                             quit), REQUIRED;
    last-work-date           "Last day of employment"
                             : DATE, REQUIRED;
    hire-date                "Original hire-date"
                             : DATE, REQUIRED;
    salary                   "Salary as of termination"
                             : Dollar-type, REQUIRED;
    );
```

%% Project %%%

```
TYPE Project-no-type        "Format for project-no values"
    = INTEGER (1..99999);
```

```
TYPE Days                   "Days of the week"
    = SYMBOLIC (sunday, monday, tuesday,
    wednesday, thursday, friday, saturday)
    ORDERED;
```



```

TYPE Week-days          "Monday through Friday"
                        = Days (monday..friday);

CLASS Project           "Current and completed Projects"
  CLASS-ATTRIBUTES
    (next-project-no    "Project-no for next new Project"
      : Project-no-type;
    )
  (project-status       "Whether current or completed"
    : SUBROLE (Current-Project,
               Previous-Project);
  project-no            "Unique project identification"
    : Project-no-type, REQUIRED, UNIQUE;
  project-manager       "Current project manager"
    : Manager, INVERSE IS projects-managing;
  project-title         "Code name for project"
    : STRING [20], REQUIRED;
  start-date            "Date project was started"
    : DATE;
  review-day            "Day of week for project reviews"
    : Week-days, REQUIRED, INITIALVALUE monday;
  review-time           "Project review meeting start"
    : TIME (07:00..17:00), REQUIRED,
      INITIALVALUE 08:00;
  dept-assigned         "Responsible department"
    : Department, REQUIRED;
  sub-projects          "Component projects, if any"
    : Project, MV, INVERSE IS sub-project-of;
  sub-project-of        "Master project, if any"
    : Project, INVERSE IS sub-projects;
);

VERIFY Valid-project-no ON Project:
  project-no LSS next-project-no AND next-project-no EXISTS
  ELSE "Project-no must be less than next-project-no";

VERIFY Valid-sub-projects ON Project:
  Project NEQ sub-projects
  ELSE "Project can not be its own sub-project";

INDEX Project-by-title ON Project
  (ASCENDING project-title, DESCENDING start-date);

INDEX Review-schedule ON Project
  (review-day, review-time);

```

%% Current-Project %%%

```
SUBCLASS Current-Project    "Projects currently in progress"
  OF Project
    (project-members        "Current employees on project"
      : Project-Employee, MV (DISTINCT, MAX 20),
      INVERSE IS current-projects;
    project-active          "Whether project has been started"
      : BOOLEAN, REQUIRED, INITIALVALUE FALSE;
  );
```

```
VERIFY Valid-start-date ON Current-Project:
  NOT project-active OR start-date EXISTS
  ELSE "Start-date must be assigned when project is active";
```

```
VERIFY Valid-project-team-minimum ON Current-Project:
  NOT project-active OR COUNT (project-members) GEQ 2
  ELSE "Active projects must have at least 2 project-members";
```

%% Previous-Project %%%

```
SUBCLASS Previous-Project   "Completed Projects"
  OF Project
    (end-date               "Date project completed"
      : DATE, REQUIRED;
    est-person-hours        "Estimated hours to complete"
      : REAL (0..1E6), REQUIRED;
  );
```

```
VERIFY Valid-end-date ON Previous-Project:
  end-date LEQ start-date
  ELSE "End-date must be less than or equal to start-date";
```

%% Department %%%

```
CLASS Department            "Departments within the company"
  (dept-no                  "Corporate department number"
    : INTEGER, REQUIRED, UNIQUE;
  dept-managers             "Managers for this department"
    : Manager, MV, INVERSE IS manager-dept;
  dept-name                 "Corporate department name"
    : STRING [20], REQUIRED;
  dept-phone                "Main office phone for department"
    : Phone-number-type, REQUIRED;
  );
```

```
VERIFY Valid-project-load ON Department:
  COUNT (projects-managing OF dept-managers) LEQ 100
  ELSE "Total projects within a department cannot exceed 100";
```

```
INDEX Department-by-name ON Department
  (dept-name);
```

### %%%%%%%% Security Information %%%%%%%%%

ACCESS Database-inquiry-access      "Inquiry access to entire database"  
     ON ALLDB  
     RETRIEVE;

ACCESS Database-modify-access      "Modify access to entire database"  
     ON ALLDB  
     RETRIEVE, MODIFY;

ACCESS Database-update-access      "Update access to entire database"  
     ON ALLDB  
     ALLDML;

ACCESS Person-public-access      "Public Person information"  
     ON Person  
     (employee-status, person-id, name, gender, spouse)  
     RETRIEVE;

ACCESS Person-retrieve-access      "Limited Person retrieve access"  
     ON Person  
     RETRIEVE  
     WHERE person-id GTR 1000;

ACCESS Employee-class-att-access      "Public Employee class attributes"  
     ON CLASS-ATTRIBUTES OF Employee  
     (next-employee-id, last-hire-date)  
     RETRIEVE;

ACCESS Employee-public-access      "Public Employee information"  
     ON Employee  
     (employee-id, employee-manager)  
     RETRIEVE;

ACCESS Employee-update-access      "Restricted Employee update access"  
     ON Employee  
     (employee-id, employee-manager)  
     ALLDML  
     WHERE exemption-status EQL non-exempt OR salary LSS 25000;

ACCESS Manager-public-access      "Public Manager information"  
     ON Manager  
     (employees-managing, projects-managing, manager-dept)  
     RETRIEVE  
     WHERE manager-level LSS executive;

ACCESS Project-public-access      "Public Project information"  
     ON Project  
     (project-no, project-manager, project-title, dept-assigned)  
     RETRIEVE  
     WHERE project-no LSS 5000;

## Basic ODL Considerations

---

```
ACCESS Department-public-access      "Public Department information"
ON Department
RETRIEVE;
```

```
PERMISSION Database-maintenance      "Maintenance analyst users"
USERCODE = (ORGDBANALYST, FINMASTER),
ACCESS   = Database-inquiry-access;
```

```
PERMISSION Database-production       "Production processing programs"
USERCODE = (PROD),
PROGRAM  = (*OBJECT/ORGDB/UPDATE ON PACK
            ,*OBJECT/ORGDB/INQUIRY ON PACK
            ),
ACCESS   = Database-update-access;
```

```
PERMISSION Public-information         "Information available to all"
USERCODE = ALL,
ACCESS   = Person-public-access, Employee-class-att-access,
            Employee-public-access, Manager-public-access,
            Project-public-access, Department-public-access;
```

%%%%%%%%%% Physical Information %%%%%%%%%%

```
DMSII-OPTIONS
(ACCESSROUTINES = *SYSTEM/ACCESSROUTINES ON DMPACK;
DMSUPPORT       = (ORGDB)DMSUPPORT/ORGANIZATION ON DMPACK;
DEFAULTS
(PACK           = DMPACK
,SECURITYGUARD = (ORGDB)G/ORGDB/STRUCTURES ON DMPACK
,DATA SET(PACK = DSPACK)
,SET          (PACK = SETPACK)
);
OPTIONS
(STATISTICS
);
PARAMETERS
(ALLOWEDCORE    = 200000
,CONTROLPOINT  = 10 SYNCPOINTS
,SYNCPOINT     = 10 TRANSACTIONS
,SYNCWAIT      = 15 SECONDS
);
AUDIT TRAIL ATTRIBUTES
(AREAS         = 50
,PACKNAME      = AUDITPACK
,ALTERNATE IS  TAPE (DENSITY = BPI6250)
,COPY TO       TAPE (3000, DENSITY = BPI6250) 1 TIMES AND REMOVE
                JOB *WFL/COPY/AUDIT ON DMPACK
,SECURITYGUARD = (ORGDB)G/ORGDB/AUDIT ON DMPACK
);
```

```
CONTROL FILE ATTRIBUTES
  (USERCODE      = ORGDB
   ,PACKNAME     = CFPACK
  );
DATABASE ORGANIZATION
  (GUARDFILE     = (ORGDB)G/ORGDB/OPENACCESS ON DMPACK
  );
CLASS Person
  (SURROGATE     = person-id;
   DATASET-OPTIONS (POPULATION = 50000);
  );
INDEX Person-by-id
  (SET-OPTIONS
   (TYPE         = INDEX-RANDOM;
    MODULUS      = 223;
    TABLESIZE   = 217;
   );
  );
SUBCLASS Employee
  (ATTRIBUTE employee-manager (MAPPING = FOREIGN-KEY);
  );
CLASS Project
  (ATTRIBUTE project-manager (MAPPING = FOREIGN-KEY);
   ATTRIBUTE dept-assigned   (MAPPING = FOREIGN-KEY);
   ATTRIBUTE sub-projects     (MAPPING = FOREIGN-KEY);
  );
);

DICTIONARY-OPTIONS
  (DICTIONARY = PRODDICTIONARY;
   DIRECTORY  = ORGDB;
  );
```



# Section 4

## Types

This section describes the SIM concept of types and the use of ODL to define and apply types. A general commentary is presented in which types are distinguished from classes. Each basic and specific kind of data type provided by SIM is described, and information on defining user-defined types is presented.

### Distinction between Types and Classes

All information stored in a SIM database consists of entities. However, SIM distinguishes between two kinds of entities:

- Dynamic entities, such as people, projects, and departments
- Data values, such as integers, dates, and strings

Some key differences between dynamic entities and data values are summarized as follows:

- Dynamic entities are stored in classes. You can explicitly insert dynamic entities and delete them from their classes through the use of OML statements. On the other hand, data values are held in types. All possible data values for types exist automatically. You need not explicitly create data values.
- Data values are considered printable because they possess well-known semantics on how they are represented graphically. For example, conventions exist to print such data values as integers and dates. In contrast, dynamic entities do not possess these predefined semantics and are not considered printable.
- Class attributes and data-valued attributes (DVAs) hold data values. Therefore, when you declare one of these attributes, you must specify a data type to indicate the type of data values that the attribute can possess. On the other hand, entity-valued attributes (EVAs) hold references to dynamic entities. Therefore, when you declare an EVA, you must specify a class to define the kind of entities that the EVA is allowed to reference.

In the remainder of this guide, the term *entity* is used to denote a dynamic entity, and the term *data type* is used to denote a type.

## Basic Kinds of Data Types

SIM provides two basic kinds of data types:

- Primitive types
- Constructor types

Primitive types, such as INTEGER or DATE, are predeclared data types that you can specify in ODL declarations.

The constructor types, which are compound and SYMBOLIC, allow you to define new data types.

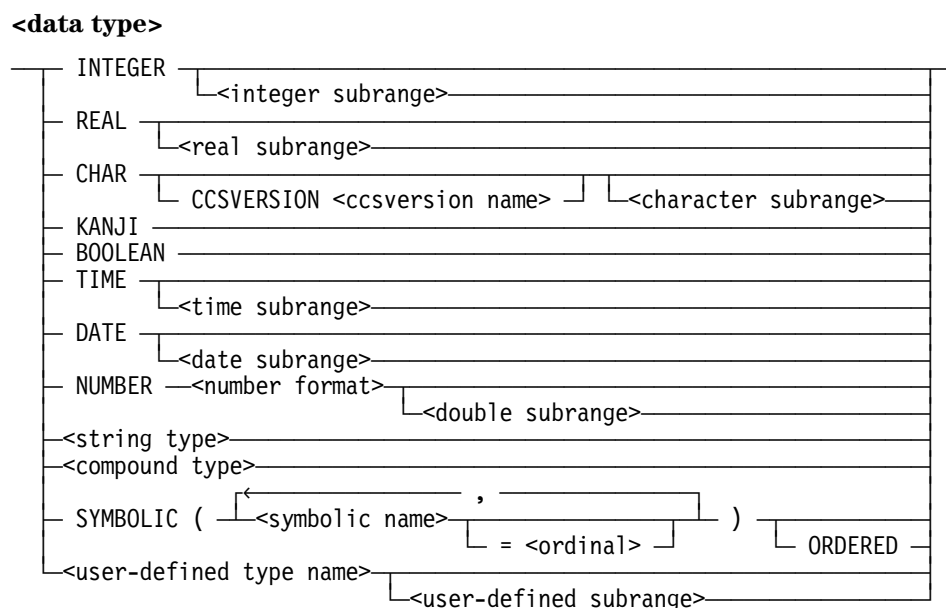
In addition to primitive and constructor types, SIM offers an ODL declaration called a user-defined type declaration. A user-defined type allows you to declare a primitive or constructor type together with added integrity constraints. You give the user-defined type a name so that it can be referenced elsewhere in the schema, such as in class attribute and DVA specifications.

## Specifying Data Types

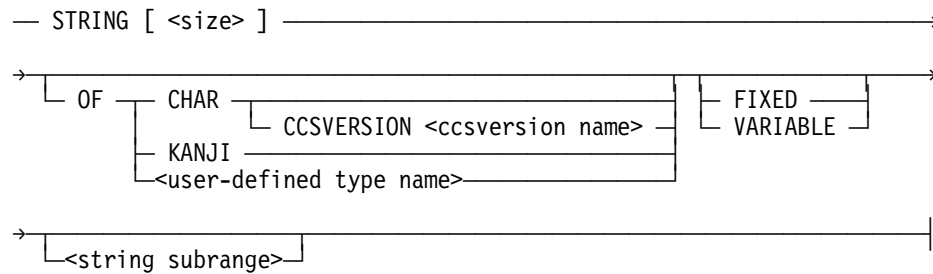
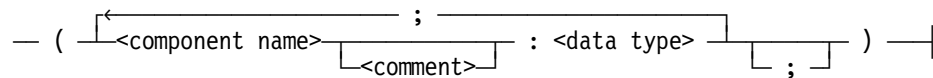
When you define user-defined types, class attributes, or DVAs, you must specify allowable data values by means of a data type specification.

### Syntax

The following diagrams show the general syntax to use for specifying a data type:





**<string type>****<compound type>**

Most of the data types that you can use in a data type specification are primitive data types (for example, `INTEGER`, `DATE`, and `STRING`). You can also use the constructor data types, namely, compound and `SYMBOLIC`. In addition, you can use a user-defined type in a data type specification. In fact, a user-defined type can be based on another user-defined type.

Each kind of data type is described in the following paragraphs.

## Primitive Types

The predeclared data types provided by SIM, called primitive types, that you can use in a data type specification are the following:

- `INTEGER`
- `REAL`
- `CHAR`
- `KANJI`
- `BOOLEAN`
- `TIME`
- `DATE`
- `NUMBER`
- `STRING`

Each primitive type represents a predefined set of data values. For example, `INTEGER` is a large set containing positive and negative integers, while `BOOLEAN` represents a set of exactly two logical values: `TRUE` and `FALSE`.

The following example shows a user-defined type, a class attribute, and a DVA whose data types are primitive types:

```
% Example user-defined type:
TYPE Phone-number-type = NUMBER [10];

CLASS Project
    % Example class attribute:
    CLASS-ATTRIBUTES
        (next-project-no :INTEGER;
        )
    % Example DVA:
    (project-title : STRING [20];
    );
```

As shown in the <data type> syntax diagram earlier in this section, most primitive type specifications can include the designation of a subrange. A subrange limits the set of allowable values to a subset of the full set of values for the primitive type.

The following example shows declarations using subranges:

```
% Example user-defined type:
TYPE Phone-number-type = NUMBER [10] (1100000000 .. 9199999999);

CLASS Project
    % Example class attribute:
    CLASS-ATTRIBUTES
        (next-project-no : INTEGER (-10 .. -1, 1 .. 99999);
        )
    % Example DVA:
    (project-title : STRING [20] ("A0" .. "C9", "ZZZZ"));
    );
```

You must observe the following constraints when you specify a subrange:

- When a subrange uses a range clause (the .. notation), the values in the range clause must be ascending. For example, the INTEGER subrange (1 .. 10) is valid, but (10 .. 1) is not valid.
- When multiple clauses are used for a subrange, the clauses can be given in any order, but they cannot overlap in values. For example, the following INTEGER subranges are valid: (11, 1 .. 10) (-1 .. 1, -2, 2 .. 3). However, the following INTEGER subranges are not valid: (5, 1 .. 10) (3, 5 .. 10, 3) (1 .. 2, 2 .. 3).

If you attempt to update an attribute through OML with a value that violates the subrange specification for that attribute, the update is rejected with an error.

Each primitive type is described in the following paragraphs. The syntax of subranges is included where applicable.

## INTEGER

The INTEGER data type represents positive and negative integer values. A description of an integer, including the range of possible values that SIM allows for it, is presented under the heading “Integer” in Section 3, “Basic ODL Considerations.” You can further restrict the values of an INTEGER data type by including an integer subrange, which has the following syntax:

### <integer subrange>

— (  $\overbrace{\text{<integer>}}^{\text{←}} \text{ , } \overbrace{\text{<integer>}}^{\text{→}} \text{ ) } \text{—————|}$   
            $\underbrace{\hspace{1.5cm}}_{\text{<integer>..<integer>}}$

The following are examples of integer subranges:

(12)  
 (-1, 1000 .. 9999, 10 .. 99)  
 (1 .. 3, 5, 7, 11, 13)

## REAL

The REAL data type represents single-precision, positive and negative, floating-point real values. A description of a real, including the range of possible values that SIM allows for it, is presented under the heading “Real” in Section 3, “Basic ODL Considerations.” You can further restrict the values of a REAL data type by including a real subrange, which has the following syntax:

### <real subrange>

— (  $\overbrace{\text{<real>}}^{\text{←}} \text{ , } \overbrace{\text{<real>}}^{\text{→}} \text{ ) } \text{—————|}$   
            $\underbrace{\hspace{1.5cm}}_{\text{<real>..<real>}}$

The following are examples of real subranges:

(0 .. 4)  
 (-1E-2 .. 0, 1 .. 99, 1E2 .. 9.9E59)  
 (-0.0001 .. 0.0001)

## CHAR

The CHAR data type represents single 8-bit characters. A description of a character is presented under the heading “Character” in Section 3, “Basic ODL Considerations.”

You can assign to an attribute based on the CHAR data type any of the 256 possible characters with internal binary values between 0 and 255. However, the ccsversion of the CHAR data type affects the behavior of the attribute in the following two ways:

- The ccsversion determines how graphically represented characters are translated into internal binary form. That is, a given graphic character might have different binary values for different ccsversions. The binary value of a character is generally only important in certain cases. For example, you might need to determine if a certain graphic character is the same as a certain character specified in hexadecimal format.
- The ccsversion also determines the collating sequence of each character. For example, “A” might be greater than “0” in one ccsversion but less than “0” in another ccsversion. Moreover, binary values of characters do not necessarily correspond to the collating sequence. For example, 4 “C1” might be considered greater than 4 “F0” in a given ccsversion. The behavior of OML logical equivalence operators (for example, EQV-EQL, EQV-GTR, and so forth) is also affected by the ccsversion.

If you do not specify a ccsversion, ASERIESNATIVE (the native ccsversion) is assumed by default. In the native ccsversion, the collating value of each character is identical to its binary value.

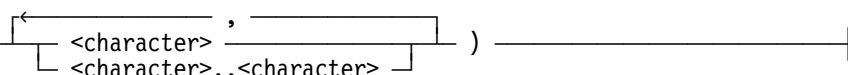
If you specify a ccsversion, the given ccsversion name must be a system identifier (letters and digits up to 17 characters in length), and it must be known to the system. When a ccsversion is defined to the system, the internal binary value and collating value of each character is specified.

The following are examples of the CHAR data type specified with a ccsversion:

- CHAR CCSVERSION FRENCH01
- CHAR CCSVERSION ARABIC

You can restrict the values that are allowed in a CHAR data type by including a character subrange, which has the following syntax:

**<character subrange>**

— (  ) —

The following are examples of character subranges:

```
("A" .. "Z", "a" .. "z", " ", "-")
("0" .. "9", 4"FF")
("!", "@", "#", "$", "%", " ")
```

Note that a character subrange bases its ordering, lowest to highest, on the collation sequence of the underlying ccsversion. Since ASERIESNATIVE bases its collation on the

EBCDIC code values of the characters, the range “A” .. “Z” includes not only the letters, but those characters represented by the EBCDIC codes that fall between the EBCDIC code values for the letters, such as “{”. The following is an example of an ASERIESNATIVE-based character subrange that includes only uppercase letters:

("A" .. "I", "J" .. "R", "S" .. "Z")

## KANJI

The KANJI data type represents all single Kanji (16-bit) characters. Kanji is the Japanese character set. A description of a Kanji is presented under the heading “Kanji” in Section 3, “Basic ODL Considerations.”

Kanji values are not ordered. Therefore, subranges are not allowed on the KANJI data type.

## BOOLEAN

The BOOLEAN data type represents the logical value TRUE or FALSE. A description of a Boolean is presented under the heading “Boolean” in Section 3, “Basic ODL Considerations.”

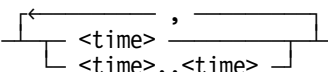
Boolean values are not ordered. Therefore, subranges are not allowed on the BOOLEAN data type.

## TIME

The TIME data type represents time data values. A description of a time, including the range of possible values that SIM allows for it, is presented under the heading “Time” in Section 3, “Basic ODL Considerations.”

You can further restrict the values of a TIME data type by including a time subrange, which has the following syntax:

**<time subrange>**

— (  ) —————

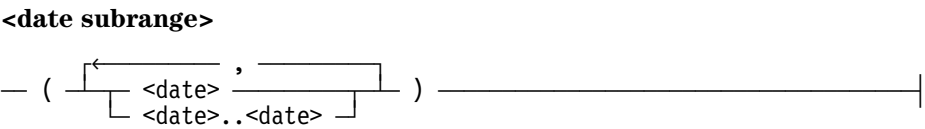
The following are examples of time subranges:

(00:00, 12:00)  
 (07:00 .. 16:59:59, 20:00:00)  
 (0:0, 1:0, 2:0, 3:0, 4:0, 5:0, 6:0, 7:0)

DATE

The DATE data type represents date data values. A description of a date, including the range of possible values that SIM allows for it, is presented under the heading “Date” in Section 3, “Basic ODL Considerations.”

You can further restrict the values of a DATE data type by including a date subrange, which has the following syntax:

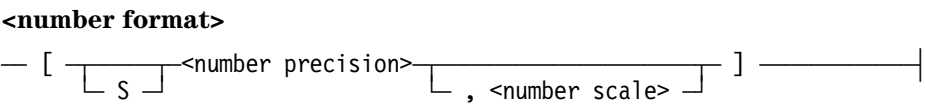


The following are example date subranges:

(2/1/90, 3/1/90, 4/1/90)  
(1/1/1901 .. 12/31/1999)  
(1/1/200 .. 12/31/300)

NUMBER

The NUMBER data type represents fixed-point integer or fractional numeric values. When you use a NUMBER data type in a <data type> specification, you define the format of the numeric values represented by the data type by specifying a number format. The number format has the following syntax:



The syntax elements shown in the diagram are explained in the following table:

| Syntax Element     | Explanation   |
|--------------------|---|
| <number precision> | Indicates how many digits of precision are represented by the NUMBER data type. A number precision is always required. The number precision must be an integer between 1 and 23, inclusive, except that if the data type is signed (S), the number precision must be between 1 and 22, inclusive. |
| S                  | Indicates that the NUMBER data type is signed and therefore represents both positive and negative values. The number precision for a signed NUMBER data type must be an integer between 1 and 22, inclusive.<br><br>If a NUMBER data type is unsigned, it represents only positive values.        |

The following are examples of NUMBER data type specifications:

When a value is assigned to a NUMBER attribute, the value is truncated on the left and rounded on the right in order to fit within the defined format for the attribute. For example, if the value -12.7 is assigned to a NUMBER [1] attribute, the value stored is 3. This action occurs because the extra left-most digits (the negative sign and the number 1) are truncated, and the extra right-most digits (.7) are rounded, causing a carryover to 3. The need for truncation or rounding is not considered an error. Therefore, no error messages are generated when this action occurs.

You can further restrict the allowable values that a NUMBER data type can have by including a subrange for the NUMBER data type. The format of this subrange is a double subrange since double-precision constant values might be needed. A double subrange has the following syntax:

( <double>, <double>..<double> )

Note that a double constant data value can have both a mantissa and an exponent greater than the maximum precision allowed for a NUMBER data type (refer to “Double” in Section 3, “Basic ODL Considerations”). Therefore, the actual mantissa and exponent that you use for a double constant within a double subrange must not specify a numeric value that exceeds the precision specified in the corresponding number format.

```
(0, 1..2, 9.9E9)
(-1.0 .. 1.0, 9.0 .. 1E20)
(-99999999999999999999..99999999999999999999)
```

The STRING data type represents strings of characters in which each character belongs to the CHAR data type or the KANJI data type. The characters are 8-bit if the STRING data type is based on the CHAR data type, and they are 16-bit if the STRING is based on the

KANJI data type. The number of characters within each string represented by a STRING data type is either fixed at precisely the number specified in the size of the STRING or it is variable between zero and the specified size, inclusive.

When a STRING data type represents 8-bit characters, its size must be an integer between 1 and 4095, inclusive. When a STRING represents 16-bit characters, its size must be an integer between 1 and 2047, inclusive.

A STRING data type specified without an OF clause is identical to a STRING specified using an OF CHAR clause without a ccsversion specification. Such a STRING is based on 8-bit characters, and its ccsversion defaults to ASERIESNATIVE (the native ccsversion). Examples of this kind of STRING data type are the following:

```
STRING [10]
STRING [20] OF CHAR VARIABLE
STRING [15] OF CHAR FIXED
```

If you specify a STRING data type with a CHAR data type and a ccsversion, the given ccsversion name must be a system identifier (letters and digits up to 17 characters in length), and it must be known to the system. In this case, each string represented by the STRING data type behaves according to the specifications of the given ccsversion. As previously described in this section under “CHAR,” the ccsversion affects how graphic characters are represented in binary form, and it affects the collating sequence of each character. The following are examples of STRING data types based on a CHAR data type specifying a ccsversion:

```
STRING [10] OF CHAR CCSVERSION FRENCH
STRING [20] OF CHAR CCSVERSION ARABIC VARIABLE
STRING [15] OF CHAR CCSVERSION ASERIESNATIVE FIXED
```

If the STRING data is based on the KANJI data type, the STRING data type represents strings of 16-bit Kanji characters. Kanji is the Japanese character set. The following is an example STRING data type based on KANJI:

```
STRING [20] OF KANJI
```



You can specify the character size and ccsversion of a STRING data type by using the OF <user-defined type name> syntax. In this case, the specified user-defined type name must be for a user-defined type based on either the CHAR or KANJI data type. The user-defined type determines the character size and ccsversion of the STRING, and it can limit the allowed character set if a character subrange is specified on the user-defined type. The following example shows this technique:

```
TYPE Name-chars = CHAR ("A" .. "Z", "a" .. "z", " ", ".");
CLASS Person
  (name      : STRING [20] OF Name-chars;
   );
```

For more information on user-defined types, refer to “User-Defined Type Declaration” later in this section.

If neither FIXED nor VARIABLE is specified in a STRING data type specification, FIXED is assumed. This option means that the STRING data type represents fixed-length strings with exactly the number of characters specified for the STRING size. If an attribute based on a fixed-length STRING is assigned a string constant that is shorter in length than the STRING size, the string constant is extended with blanks on the right so that its length is that of the STRING size. This process is called blank padding.

If a STRING data type is specified as VARIABLE, it represents all possible strings with lengths between 0 and the specified size, which constitutes the maximum string length. An attribute based on a variable-length STRING possesses an implicit length property that indicates the number of characters actually assigned to the attribute the last time it was modified. For example, consider the following attribute specification:

```
S: STRING [10] VARIABLE
```

If S is assigned the value ABC, blank padding is not performed and the OML expression LENGTH(S) returns the value 3. In contrast, if S were specified as FIXED, the constant ABC would be blank padded with seven blank characters and LENGTH(S) would always return the value 10.

Note that an attribute based on a variable-length string can be assigned an empty string (""), thereby giving it a length of zero. Also, the use of the VARIABLE option does not affect the number of characters actually stored on disk for a STRING attribute. Refer to Section 9, “DMSII Mapping Options,” for information on compaction options.

For a STRING data type based on a CHAR data type, you can restrict the strings represented by the data type by including a string subrange, which has the following syntax:

**<string subrange>**

```
— ( [ <string> , <string> ] | <string>..<string> ) —————
```

The following are examples of string subranges:

```
("A0000".."Z9999")
("A".."Z", "a".."z", "0".."9", " ", 4"FFFFFF")
("Antarctica", "Asia", "North America")
```

Note that for a STRING data type based on a user-defined type, all string constants used within a string subrange must adhere to any subrange restrictions defined for the user-defined type. For example, if the user-defined type does not allow the character A, the subrange specification for the STRING data type could not use the string constant ABC.

## Constructor Types

In addition to the primitive types, there are two constructor types that you can use to define new data types:

- Compound
- SYMBOLIC

Unlike primitive types, which are predefined and merely referenced throughout a schema, each constructor type creates a type when it is defined. Furthermore, the set of possible data values that belong to a constructor type are implicitly created when the type is defined.

The following are examples of declarations using constructor types:

```
TYPE Address-type      "Address-type is a compound user-defined type" =
  (street              : STRING [30];
   city                : STRING [20];
   state               : STRING [2];
   zipcode             : STRING [9];
  );

SUBCLASS Employee OF Person
  (exemption-status    "exemption-status is a SYMBOLIC DVA" :
    SYMBOLIC (exempt, non-exempt);
   work-phone          "work-phone is a compound DVA" :
    (area-code         : NUMBER [3];
     prefix            : NUMBER [3];
     extension         : NUMBER [4];
    );
  );
```

Each constructor data type is described in the following paragraphs.

## Compound

The compound data type defines a set of one or more components, each of which has a name and a data type. For each instance of the compound, each component can hold a single value corresponding to its data type or it can be null. Since the compound consists solely of data-valued components, the compound itself is considered a printable data value. The compound data type allows you to define a data type consisting of arbitrarily complex data values.

The following example shows a DVA with a compound data type:

```
CLASS Person
  (home-phone-number :
    (area-code       : NUMBER [3];
     prefix          : NUMBER [3];
     suffix          : NUMBER [4];
    )
  )
```

Each component name at a given level within a compound must be unique. For instance, in the preceding example, area-code could not be used twice within the home-phone-number attribute. However, the same component name can be used at different levels within a compound. For example, if the prefix component in the preceding example were itself a compound, it could contain a component named area-code.

When defining a SIM schema, the question often arises as to whether to use a compound DVA or an EVA referencing a separate class. For example, in the preceding example the home-phone-number attribute could have been defined as an EVA pointing to a class named Phone-number, as follows:

```
CLASS Person
  (home-phone-number : Phone-number
  );

CLASS Phone-number
  (area-code       : NUMBER [3];
   prefix          : NUMBER [3];
   suffix          : NUMBER [4];
  );
```

The use of compounds for modeling complex data values has certain advantages, while the use of EVAs and separate classes has other advantages.

Some of the advantages of using compounds instead of EVAs and separate classes are the following:

- A compound attribute can be assigned a value when an entity in the corresponding class is inserted. For example, if home-phone-number is a compound DVA, it can be assigned a value in the same OML statement that inserts a Person entity. If home-phone-number is an EVA, the Phone-number entity must be explicitly inserted through a separate OML statement before it can be referenced by a Person entity.
- Comparisons between compound attributes are value-based. For example, if the home-phone-number of two Person entities are compared through the equals (=) operator, SIM actually compares each component of one entity to its counterpart in the other entity. If all matching components between the two entities are nonnull and have the same data value, the compounds are considered equal. In contrast, when home-phone-number EVAs are compared, they are not considered equal unless they reference the identical entity, regardless of the DVA values of each entity.
- Since compound attributes are printable, their values are easy to display. For example, the home-phone-number of each Person can be displayed through the OML query *FROM Person RET home-phone-number*. If home-phone-number is an EVA, the same query returns a reference (surrogate value) to each Phone-number value, and this reference is generally not printable. To retrieve a printable value for home-phone-number, explicit DVA values must be retrieved from the Phone-number class. The retrieval query might be *FROM Person RET \* OF home-phone-number* or *FROM Person RET (area-code, prefix, suffix) OF home-phone-number*.

Some of the advantages of using EVAs and separate classes instead of compounds are the following:

- Compounds can only contain components that are data-valued. Therefore, a compound cannot contain an EVA that references an entity. Furthermore, compound components are always single valued, and they cannot possess attribute options such as REQUIRED or INITIALVALUE. Even if a compound attribute does not require any of these features initially, it might require some of these features at a later time. Consequently, a class can be preferable over a compound in that EVAs, attribute options, and even subclasses can always be added to a class through schema changes.
- A class can be used as a perspective in a query, whereas a compound cannot be used in this manner. For example, if Phone-number is a class, you can ask for all phone numbers used by more than one Person entity through the query *FROM Phone-number RET \* WHERE COUNT (INVERSE (home-phone-number)) > 1*. If home-phone-number were a compound, the same query would be considerably more difficult to state.
- Representing complex data values as entities eliminates redundant data, thereby simplifying updates and possibly reducing disk space requirements and improving performance. For example, if each phone number exists precisely once as a Phone-number entity, the area code of a single phone number can be changed by updating a single entity. In contrast, if phone numbers were represented as compounds, many Person entities might have to be updated to reflect the same change.

The advantages of compounds must be weighed against the advantages of the more general EVA/class approach when considering either approach for a particular database schema.

## SYMBOLIC

The SYMBOLIC data type is similar in concept to the programming language concept of enumerated types. The SYMBOLIC data type allows you to define a set of data values that have a symbolic name and, optionally, an ordinal value. Since it is a constructor data type, the SYMBOLIC data type defines both a new data type and the symbolic data values that belong to that type. The following are examples of declarations that use the SYMBOLIC data type:

```
TYPE Degree-type      = SYMBOLIC (hs, ba, bs, ma, ms, phd) ORDERED;
CLASS Person
  (marital-status      : SYMBOLIC (single, married, divorced);
   terminate-reason    : SYMBOLIC (retired = 4, death = 9,
                                   quit = 57, fired = 20);
  );
```

The symbolic name used to identify each SYMBOLIC data value must be a valid SIM identifier. The symbolic names are used both in attribute assignments and in comparisons of attributes that are based on a SYMBOLIC data type. In effect, each symbolic name becomes a named constant data value; consequently, the corresponding SIM identifier cannot be used in another declaration in the same database.

As shown in the preceding example, a SYMBOLIC data type can be ORDERED. When you specify ORDERED, the SYMBOLIC data values are considered to be ordered and their precedence is given by the order in which they are specified. For example, in the user-defined type Degree, *hs* is considered less than *ba*, which is less than *bs*, and so forth. You can use ordered SYMBOLIC data values in the OML PRED and SUCC functions. Also, you can use attributes that are based on ordered SYMBOLIC data types in inequality comparisons, such as GTR (>).

By default, a SYMBOLIC data type is considered unordered; unordered SYMBOLIC data values cannot be used in the OML PRED and SUCC functions. Furthermore, attributes based on an unordered SYMBOLIC can only be used for equality (= or EQL) and strict inequality (<> or NEQ).

By default, SIM assigns each SYMBOLIC data value an integer that is unique among all data values of the same type. This ordinal value begins at zero and is increased by one for each SYMBOLIC data value in the order given.

Optionally, you can explicitly assign an ordinal value to a SYMBOLIC data value by using the `= <ordinal>` syntax. When this syntax is used, the following conventions apply:

- The ordinal value must be a value between 0 and 268435455 ( $2^{28} - 1$ ), inclusive.
- If a SYMBOLIC data value is not assigned an explicit ordinal value, SIM implicitly assigns it an ordinal value one greater than the ordinal value of the previous SYMBOLIC data value. Zero is the implicit ordinal number of the first data value unless an explicit value is given.
- The same ordinal value cannot be used twice for the same SYMBOLIC data type.
- If a SYMBOLIC data type is ORDERED, the ordinals of its data values must be ascending, and there must be no unused values between its lowest and highest ordinal values.

Explicitly assigned ordinal values can be used, for example, to match numeric constant values that are controlled outside of the database.

## User-Defined Type Name

In addition to using primitive and constructor types, you can use any user-defined type as a data type. As shown in the `<data type>` syntax diagram earlier in this section, you do so by specifying the user-defined type name, optionally followed by a user-defined subrange, as a data type.

The following conventions apply when you specify a user-defined type as a data type:

- A user-defined type must be declared before it can be referenced in a data type specification. Therefore, the given user-defined type name must correspond to an already defined user-defined type.
- The data type specification assumes the data type on which the given user-defined type is based. For example, a user-defined type that is based on the INTEGER data type is itself considered an INTEGER data type.
- A user-defined type based on the CHAR or STRING OF CHAR data type assumes the ccsversion of the underlying CHAR or STRING. Anything based on that user-defined type also inherits the relevant ccsversion.
- If the user-defined type invoked any subrange restrictions in its declaration, those restrictions apply to anything based on that user-defined type. In other words, a user-defined type causes its subrange restrictions to be inherited by anything that is based on it.



# User-Defined Types

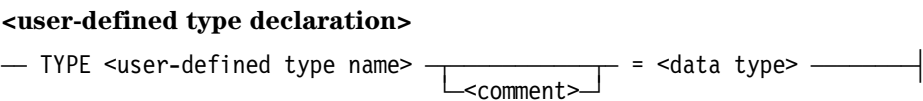
A user-defined type can be used for two basic purposes. First, you can use it to attach a name and optionally a subrange specification to a data type. The user-defined type can then be referenced in multiple places in the schema, thereby eliminating the need to replicate the subrange restrictions in each place. Second, you can use a user-defined type to define and attach a name to a constructor type so that it can be used by multiple attributes within the schema.

## User-Defined Type Declaration

Information on defining a user-defined type through a user-defined type declaration is presented in the following paragraphs.

### Syntax

The following diagram shows the general syntax to use for defining a user-defined type:



### Explanation

The syntax elements shown in the diagram are explained in the following table:

| Syntax Element                | Explanation  |
|-------------------------------|--|
| TYPE <user-defined type name> | Identifies uniquely the name of the type that you are defining. The name can be any combination of letters, digits, hyphens, or underscores, starting with a letter and ending with a letter or digit, up to 30 characters in length.  |
| <comment>                     | Denotes optional remarks about this user-defined type, such as a brief description of its purpose. This comment is to be included in the schema file and saved in the SIM database directory. The comment must be enclosed in quotation marks ( " ").  |
| <data type>                   | <p>Identifies the data type for this user-defined type. The data type may be a primitive type, a constructor type, or another user-defined type.</p> <p>If the data type is a user-defined type, it must be previously defined in the schema.</p> <p>The data type specification may include a subrange of values.</p> |



**Syntax Element****Explanation**

Information on specifying a data type and descriptions of the primitive types and constructor types are presented under the heading “Specifying Data Types” earlier in this section.

## Examples of User-Defined Type Declarations

Examples of user-defined type declarations are presented in the following paragraphs.

**Example 1**

This example demonstrates the use of a user-defined type to eliminate redundant subrange specifications.

Consider the following schema:

```
CLASS Project
  CLASS-ATTRIBUTES
    (next-project-no : INTEGER (1 .. 99999);
    )
    (project-no      : INTEGER (1 .. 99999);
    );
```

In this schema, both the next-project-no class attribute and the project-no DVA are based on the INTEGER data type with the same subrange restrictions. With this approach, each time an attribute is declared that can hold a project number, the same data type specification must be made. A user-defined type can be used to eliminate these redundant specifications, as follows:

```
TYPE Project-no-type = INTEGER (1 .. 99999);

CLASS Project
  CLASS-ATTRIBUTES
    (next-project-no : Project-no-type;
    )
    (project-no      : Project-no-type;
    );
```

Besides eliminating redundancy, this use of user-defined types has the advantage of providing a single point of definition. That is, if the semantics of a project number were changed, a single schema declaration could be changed and all applicable attributes would automatically reflect that change.

**Example 2**

Consider a schema in which a constructor data type is used in-line, that is, without using a user-defined type:

```
CLASS Project
  (review-day : SYMBOLIC (monday, tuesday, wednesday, thursday,
```

```
        friday);  
    );
```

In this schema, Project has a DVA based on the SYMBOLIC data type that represents the day of the week on which each project is reviewed. Suppose that a requirement were to arise to add a multivalued attribute to Project to indicate which days of the week are authorized for overtime. Optimally, the attribute should also be a SYMBOLIC data type that uses the same SYMBOLIC data values. However, since SYMBOLIC data types are constructor types, a new in-line SYMBOLIC data type definition would define a new type. The new SYMBOLIC data type could not, for example, reuse the SYMBOLIC data values used by the first SYMBOLIC data type.

This scenario demonstrates the fact that when constructor types are defined in-line, they are private in a sense and cannot be used by multiple attributes. A user-defined type can resolve this situation, as shown in the following changed schema:

```
TYPE Week-day      = SYMBOLIC (monday, tuesday, wednesday, thursday,  
                               friday);  
CLASS Project  
    (review-day    : Week-day;  
     overtime-days : Week-day, MV;  
    );
```

As shown, the SYMBOLIC data type is defined once as the data type of a user-defined type, Week-day. The user-defined type essentially attaches a name to the SYMBOLIC data type so that it can be used by multiple attributes within the schema. As with Example 1, the user-defined type also serves as a single point of definition, allowing the semantics of a Week-day type to be changed in a single schema declaration.

**Example 3**

The following example schema shows a variety of typical cases in which a user-defined type can be used to eliminate redundancy and declare constructor types such that they can be shared by multiple attributes. This example also demonstrates the building of a user-defined type from another user-defined type.

```

TYPE Phone-number-type =
  (area-code      : NUMBER [3];
   prefix         : NUMBER [3];
   suffix         : NUMBER [4];
  );

TYPE Days          = SYMBOLIC (sunday, monday, tuesday,
                               wednesday, thursday, friday, saturday)
                               ORDERED;

TYPE Week-days     = Days (monday .. friday);

TYPE Early-week-days = Week-days (monday .. wednesday);

TYPE Week-end-days = Days (sunday, saturday);

CLASS Person
  (home-phone-number : Phone-number-type;
   office-phone-number : Phone-number-type;
  );

CLASS Project
  (work-days      : Days, MV (MAX 5);
   review-day     : Early-week-days;
   overtime-days  : Week-end-days, MV;
  );

```

In this example, a compound user-defined type named `Phone-number-type` is declared, and it is used for two DVAs of `Person`: `home-phone-number` and `office-phone-number`. This approach allows the compound to be defined once, and it allows the two DVAs to have exactly the same data type. This approach allows the attributes to be compared to each other.

The `SYMBOLIC` user-defined type `Days` defines all seven days of the week. The user-defined type `Week-days` is then based on `Days`, restricting the possible data values to only Monday through Friday. `Early-week-days` defines a further-restricted set of only Monday through Wednesday. `Week-end` is based on `Days` and allows only Sunday and Saturday as its data values. The `Project` class has three DVAs that are based on these user-defined types. Although these DVAs have different allowed data values, they are all ultimately based on the same `SYMBOLIC` data type. Consequently, they can be assigned and compared to one another.



# Section 5

## Classes

This section describes the use of ODL to define classes. All examples presented in this section reflect the ORGANIZATION database, which is described in Section 3, “Basic ODL Considerations.”

A class is a collection of entities of the same basic type. These entities are objects of interest that have characteristics in common. For example, the entities represented by the Person class are people.

When you define a class, you define all the attributes that characterize the entities of that class. These attributes can be data-valued attributes (DVAs) or entity-valued attributes (EVAs). A DVA holds printable values, such as a number, a date, or a string. For example, the DVA salary holds the salary of each employee. An EVA relates an entity to one or more entities of the same or another class. For example, the EVA employees-managing relates a manager to his or her employees.

SIM assigns a unique surrogate value to each entity. This surrogate value allows SIM to distinguish the entity from all other entities regardless of its attribute values.

When you define a class, you can also define class attributes for the class. It is important to distinguish between attributes and class attributes. Whereas an attribute (namely, a DVA or an EVA) holds one or more values for each entity, a class attribute holds one value for the entire class. For example, the class attribute next-project-no in the Project class holds a number that identifies a project.

It is possible for a class to be defined with no attributes; when this occurs, each entity has only a surrogate value.

Every SIM database must have at least one class. The optimum number of classes in a SIM database depends on the nature and complexity of the particular database application.

# General Concepts

Some general concepts that pertain to classes are presented in the following paragraphs; refer to Figure 3–1 for a graphic representation of the relationships between classes described in the examples.

## Entity Relationships

In most applications, each entity typically has many relationships with other entities in the database. An important advantage of SIM is that these entity relationships can be defined in the schema. SIM is thereby allowed to employ the semantics of the relationships for integrity and query purposes.

With SIM, an entity relationship is defined with a pair of EVAs. Each EVA defines the relationship from one perspective, namely, the perspective of the class in which the EVA is defined. For example, the EVA project-manager defines a relationship between the Project class and the Manager class from the perspective of the Project class. Projects-managing is the inverse EVA that defines the relationship from the perspective of the Manager class. Each EVA can have attribute options that specify integrity constraints that differ from those of its inverse EVA.

If desired, only one EVA need be defined to establish an entity relationship. In this case, SIM automatically declares an unnamed, implicit inverse EVA with default attribute options. EVAs and entity relationships are described in detail under the heading “Entity-Valued Attributes (EVAs)” later in this section.

## Generalization Hierarchies

SIM has the powerful ability to arrange classes in generalization hierarchies. A generalization hierarchy, or, simply, a “hierarchy,” is a collection of classes that define roles in which entities can participate. Every entity is a member of exactly one hierarchy.

Each hierarchy has one base class, in which all entities of that hierarchy participate. A base class defines attributes that apply to all entities in the hierarchy. For example, Person is a base class; therefore, all entities in the corresponding hierarchy have a name, a birth-date, and so on.

All classes in the hierarchy other than the base class are called subclasses. The attributes of a subclass apply only to entities that participate in that subclass. For example, the Employee subclass defines such attributes as employee-id and salary, but only Person entities that participate in the Employee subclass have values for these attributes. Note that an entity that participates in both the Person base class and the Employee subclass is not two separate entities; it is a single entity that has two roles.

A subclass can have subclasses of its own. For example, Project-Employee is a subclass of Employee. When referenced as a superior of a subclass, a base class or subclass is called a superclass. An entity that participates in a subclass must participate in all of its superclasses. For example, a Project-Employee entity must also participate in the Employee subclass and the Person base class.

Since subclasses can have subclasses, hierarchies can arbitrarily have many levels.

Base classes and subclasses can have multiple immediate subclasses, and unless restricted by integrity constraints, an entity can participate in multiple immediate subclasses simultaneously. For example, an Employee entity can participate in the Project-Employee and Manager subclasses simultaneously.

An entity that participates in a subclass inherits all the attributes of all of its superclasses. For example, since all Employee entities must participate in the Person base class, all Employee entities have a name.

SIM allows a subclass to have multiple immediate superclasses. For example, the Interim-Manager subclass has both Manager and Project-Employee as immediate superclasses. According to hierarchy rules, this means that for an entity to participate in the Interim-Manager subclass, it must also participate in both of its immediate superclasses, namely, the Manager and Project-Employee subclasses. As is the case with a subclass that has a single immediate superclass, a subclass with multiple immediate superclasses inherits all the attributes of all of its superclasses.

Hierarchies must be arranged in a pattern known as a directed acyclic graph (DAG). This pattern means that a hierarchy must meet the following rules:

- All hierarchies must stem from a single base class. In other words, all superclass paths in a hierarchy must meet at the same base class.
- Cycles are not allowed. That is, a subclass cannot be a superclass of itself.
- All attribute names of a subclass, including inherited attributes, must be unique. For example, the Manager and Project-Employee subclasses cannot both have an attribute named salary because the Interim-Manager subclass would then inherit the same attribute name twice. However, the Employee and Previous-Employee subclasses could both have an attribute named salary because no subclass inherits the attributes of both Employee and Previous-Employee.
- The subrole DVA of a least common ancestor (LCA) must be multivalued. An LCA is the lowest point at which multiple superclass paths meet. For example, the superclass paths of Interim-Manager meet at the Employee subclass; therefore, the subrole DVA called employee-roles that is within Employee must be multivalued.

Except where noted in this guide, the term *class* refers to a base class or a subclass. The term *superclass* refers to a class that has subclasses.

## Defining Classes

From the standpoint of defining the classes of a SIM database, there are two kinds of classes: base classes and subclasses.

You define a base class by means of a base class declaration. The general syntax for a base class declaration is presented under the heading “Base Class Declaration” later in this section.

You define a subclass by means of a subclass declaration. The main difference between a base class declaration and a subclass declaration is that a superclass or superclasses must be specified in the subclass declaration. The general syntax for a subclass declaration is presented under the heading “Subclass Declaration” later in this section.

The syntax for defining class attributes and attributes in a base class declaration or a subclass declaration is presented under the headings “Class Attributes” and “Attributes” later in this section.

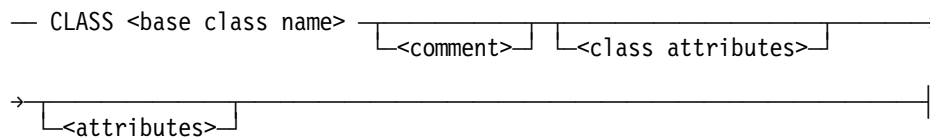
### Base Class Declaration

Information on defining a base class through a base class declaration is presented in the following paragraphs and in the paragraphs under the headings “Class Attributes” and “Attributes” later in this section. A base class is any class that has no superclasses.

#### Syntax

The following diagram shows the general syntax to use for defining a base class:

#### **<base class declaration>**





**Explanation**

The syntax elements shown in the diagram are explained in the following table:

| Syntax Element          | Explanation   |
|-------------------------|---|
| CLASS <base class name> | Identifies uniquely the name of the base class that you are defining. A base class name must be a valid SIM identifier, namely, any combination of letters, digits, hyphens, or underscores, starting with a letter and ending with a letter or digit, up to 30 characters in length. |
| <comment>               | Denotes optional remarks about the base class, such as a brief description of its purpose. This comment is to be included in the schema file and saved in the SIM database directory. The comment must be enclosed in quotation marks (" ").  |
| <class attributes>      | Specifies all class attributes associated with the base class that you are defining. The syntax for identifying class attributes is presented under the heading "Class Attributes" later in this section.   |
| <attributes>            | Specifies all attributes associated with the base class that you are defining. The syntax for identifying attributes is presented under the heading "Attributes" later in this section.   |

As described under "Generalization Hierarchies" earlier in this section, a base class forms the base of a generalization hierarchy. All entities in the hierarchy participate in the base class.

Both class attributes and attributes are optional for a base class declaration. Each class attribute holds one value for the entire class. Each attribute (a DVA or an EVA) holds one or more values for each entity in the class. A class with no attributes causes the corresponding entities to have only a surrogate value.

Note that for a base class (or a subclass) to have subclasses, it must have one attribute that is a DVA of the type SUBROLE (a subrole DVA). Refer to "Data-Valued Attributes (DVAs)" later in this section for a description of a subrole DVA.

### Example

```
CLASS Project "All projects currently in progress"  
    CLASS-ATTRIBUTES  
        (next-project-no : INTEGER;  
         )  
    (project-no : INTEGER, REQUIRED, UNIQUE;  
     project-title : STRING [20];  
     project-manager : Manager INVERSE IS projects-managing;  
     project-members : Project-Employee MV (MAX 20),  
                     INVERSE IS current-projects;  
    );
```

This example shows part of the base class declaration for the Project base class. (For the complete declaration, refer to “ORGANIZATION Database” in Section 3, “Basic ODL Considerations.”) A comment about the Project base class is contained in quotation marks. A class attribute named next-project-no is specified, as are DVAs named project-no and project-title. EVAs named project-manager and project-members are also specified, and their inverse EVAs are identified.

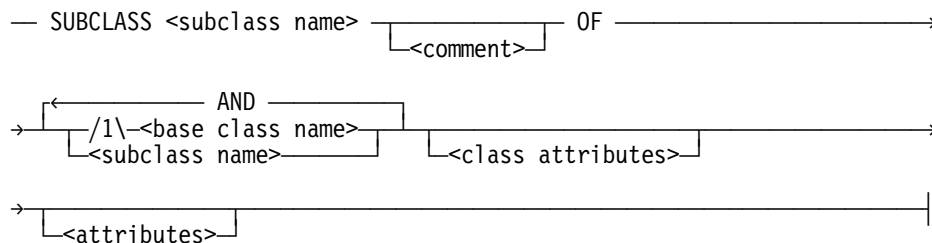
## Subclass Declaration

Information on defining a subclass through a subclass declaration is presented in the following paragraphs and in the paragraphs under the headings “Class Attributes” and “Attributes” later in this section. A subclass is any class that has one or more superclasses. A subclass can have only one superclass that is a base class.

## Syntax

The following diagram shows the general syntax to use for defining a subclass:

**<subclass declaration>**



**Explanation**

The syntax elements shown in the diagram are explained in the following table:

| Syntax Element           | Explanation   |
|--------------------------|---|
| SUBCLASS <subclass name> | Identifies uniquely the name of the subclass that you are defining. A subclass name must be a valid SIM identifier, namely, any combination of letters, digits, hyphens, or underscores, starting with a letter and ending with a letter or digit, up to 30 characters in length. |
| <comment>                | Denotes optional remarks about the subclass, such as a brief description of its purpose. This comment is to be included in the schema file and saved in the SIM database directory. The comment must be enclosed in quotation marks (" ").  |
| OF <class name>          | Identifies a base class that is a superclass of the subclass you are defining.  |
| OF <subclass name>       | Identifies a subclass that is a superclass of the subclass you are defining.  |
| AND                      | Indicates that another superclass of the subclass that you are defining is to be identified.  |
| <class attributes>       | Specifies all class attributes associated with the subclass that you are defining. The syntax for identifying class attributes is presented under the heading "Class Attributes" later in this section.   |
| <attributes>             | Specifies all attributes associated with the subclass that you are defining. The syntax for identifying attributes is presented under the heading "Attributes" later in this section.   |

As described under "Generalization Hierarchies" earlier in this section, a subclass declaration must meet certain rules to ensure that the hierarchy is a directed acyclic graph (DAG). A subclass has one or more superclasses, only one of which can be a base class. A subclass can itself have one or more subclasses. Entities in the hierarchy only participate in a subclass when they are inserted into that subclass role. A subclass inherits all class attributes and attributes of all of its superclasses. As is the case for a base class declaration, both class attributes and attributes are optional for a subclass declaration.

### Example

```

SUBCLASS Employee      "Persons who are employees"
  OF Person
  (employee-roles      : SUBROLE (Project-Employee, Manager) MV;
   employee-id         : INTEGER, UNIQUE, REQUIRED;
   hire-date           : DATE;
   salary              : REAL;
   exemption-status    : SYMBOLIC (exempt, non-exempt);
   employee-manager    : Manager INVERSE IS employees-managing;
  );
  
```

This example shows the Employee subclass declaration, which has a single superclass of Person. A comment about the Employee subclass is contained in quotation marks. No class attributes are specified for the subclass. DVAs named employee-roles, employee-id, hire-date, salary, and exemption-status are specified. An EVA named employee-manager is specified, and its inverse EVA is identified.

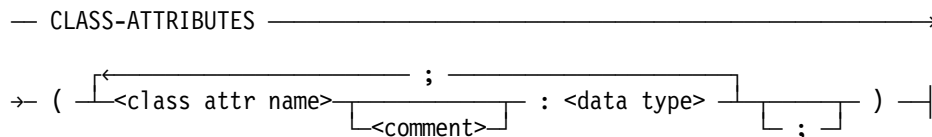
## Class Attributes

Information on defining class attributes in a base class declaration or a subclass declaration is presented in the following paragraphs. Class attributes are characteristics of a class as a whole rather than characteristics of the entities of a class. A class attribute holds one value for the entire class. For example, the class attribute next-project-no in the Project base class can be used for assigning an identification number to a new project.

### Syntax

The following diagram shows the syntax to use for defining class attributes:

#### <class attributes>



**Explanation**

The syntax elements shown in the diagram are explained in the following table:

| Syntax Element    | Explanation  |
|-------------------|--|
| CLASS-ATTRIBUTES  | Indicates that the construct or constructs that follow, within parentheses, are to be class attributes.  |
| <class attr name> | Specifies the name of a class attribute you are defining. The name must be a valid SIM identifier, namely, any combination of letters, digits, hyphens, or underscores, starting with a letter and ending with a letter or digit, up to 30 characters in length. |
| <comment>         | Denotes optional remarks about the class attribute, such as a brief description of its purpose. This comment is to be included in the schema file and saved in the SIM database directory. The comment must be enclosed in quotation marks (" ").                |
| <data type>       | Specifies the data type of the class attribute. For detailed information about the <data type>, refer to Section 4, "Types."   |

Since each class attribute has a single value for the entire database, it can be regarded as a global value. Even when a class is empty (that is, when it contains no entities), its class attributes still have values. When a class attribute is first defined, SIM gives it an initial value of NULL. You can then assign it a value other than NULL by means of an OML *MODIFY* statement. Class attributes cannot be assigned a value by means of an INSERT statement. Moreover, they cannot be modified by means of a DELETE statement. Refer to the *SIM OML Programming Guide* for more information on manipulating the values of class attributes.

**Example**

```

CLASS Project          "Current and completed Projects"
  CLASS-ATTRIBUTES
    (next-project-no   "Project-no for next new Project"
      : INTEGER;
    )
  (
    ...
  );

```

This example shows the next-project-no class attribute in the base class declaration for the Project base class. (For the complete declaration, refer to "ORGANIZATION Database" in Section 3, "Basic ODL Considerations.")

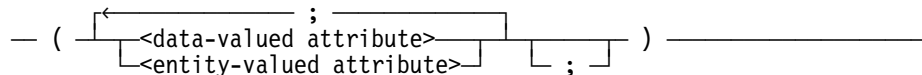
### Attributes

The attributes in a base class declaration or a subclass declaration can be data-valued attributes (DVAs) or entity-valued attributes (EVAs).

#### Syntax

The following diagram shows the general syntax to use for defining attributes:

**<attributes>**



As shown in the diagram, <data-valued attribute> and <entity-valued attribute> definitions can be specified in any order. These definitions are described under the headings “Data-Valued Attributes (DVAs)” and “Entity-Valued Attributes (EVAs).”

### Data-Valued Attributes (DVAs)

Information on defining data-valued attributes (DVAs) in a base class declaration or a subclass declaration is presented in the following paragraphs.

A DVA is an attribute that holds printable values, such as strings, numbers, or dates. A DVA holds one or more values for each entity that participates in the class in which the DVA is defined. The kinds of values that a DVA can hold depend on its type, which can be either a data type or the special type SUBROLE.

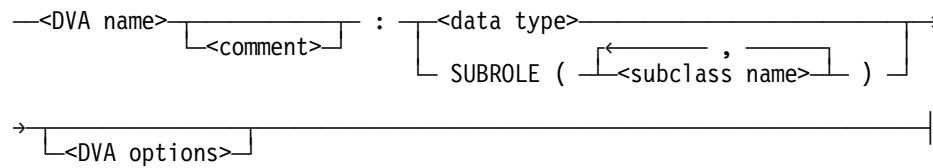
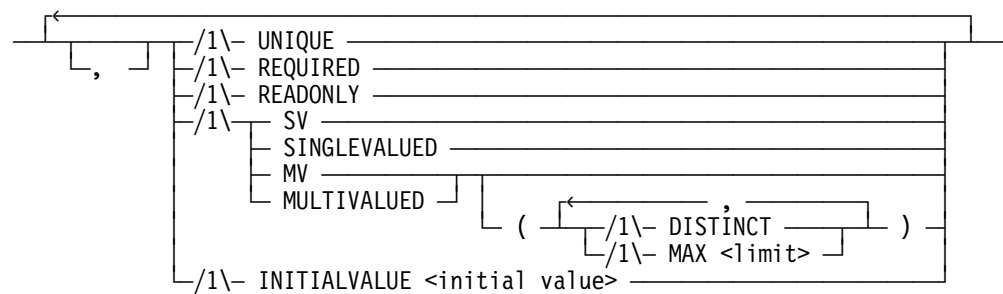
When the DVA type is a data type, the DVA is considered to be user-controlled. That is, the values for the DVA can be assigned explicitly by means of OML statements when entities are inserted and modified, or implicitly by means of an INITIALVALUE declaration.

A class can have one DVA of the type SUBROLE. This DVA is called the subrole DVA, and it indicates that the class in which it is defined has subclasses. You must declare precisely one subrole DVA for each class that is to have subclasses. The subrole DVA serves two purposes:

- It enumerates the subclasses of the class for which it is declared.
- It allows integrity constraints to be specified that affect the way entities behave within the hierarchy. Integrity constraints are discussed later in this section.

**Syntax**

The following diagrams show the syntax to use for defining a data-valued attribute (DVA):

**<data-valued attribute>****<DVA options>****<initial value>**

### Explanation

The syntax elements shown in the diagrams are explained in the following table and in the paragraphs that succeed the table.

| Syntax Element  | Explanation   |
|-----------------|---|
| <DVA name>      | Specifies the name of the DVA that you are defining. The name must be a valid SIM identifier, namely, any combination of letters, digits, hyphens, or underscores, starting with a letter and ending with a letter or digit, up to 30 characters in length. |
| <comment>       | Denotes optional remarks about the DVA, such as a description of its purpose. This comment is to be included in the schema file and saved in the SIM database directory. The comment must be enclosed in quotation marks (" ").                             |
| <data type>     | Specifies the data type of the DVA. For detailed information about the <data type> syntax, refer to Section 4, "Types."   |
| SUBROLE         | Specifies that the DVA is a read-only attribute of the type SUBROLE (a subrole DVA), and that the base class or subclass you are defining has subclasses.   |
| <subclass name> | Specifies the name of a subclass of the base class or subclass that you are defining. A subclass cannot be specified as its own subclass.   |
| <DVA options>   | Indicates characteristics of the DVA, such as integrity constraints. The DVA options are described in the paragraphs following this table.  |

### DVA Options

You optionally can specify DVA options to constrain the values that a DVA can have or to affect the behavior of the DVA in some way. When a DVA option serves as a constraint, any OML statement that attempts to violate the constraint is aborted and an error message results. If you add a constraint to an existing DVA by means of a CHANGE command, SIM ensures that all existing data meets the new constraint. (Refer to Section 12, "Changing a SIM Database Schema," for restrictions and implications with respect to changing a schema.)



Each DVA option is explained in the following paragraphs.

#### UNIQUE

The UNIQUE option ensures that each value for the DVA throughout the entities in the class is never used twice. That is, whenever the DVA is assigned a value by means of an OML *INSERT* or *MODIFY* statement, SIM ensures that the new value is not already used. The UNIQUE option applies to both single-valued and multivalued DVAs. By default, DVAs are not UNIQUE.

You cannot specify the UNIQUE option for a subrole DVA. Also, when this option is specified for a DVA that is a number, the DVA must be either signed with a precision of 11 digits or less, or unsigned with a precision of 12 digits or less. This restriction results from DMSII limitations on index keys.

#### REQUIRED

The REQUIRED option ensures that the DVA is never null. When applied to a multivalued DVA, the REQUIRED option ensures that the DVA has at least one value. Note that the INITIALVALUE option, which is described later in this section, can be used to assign a default value to a DVA when entities are inserted. By default, DVAs are not REQUIRED.

The REQUIRED option does not interact with the UNIQUE option. That is, UNIQUE does not imply REQUIRED. Therefore, a DVA that is UNIQUE but not REQUIRED is allowed to be null for an unlimited number of entities.

Note that when a subrole DVA is defined as REQUIRED, the entities must participate in at least one subclass of the class that owns the DVA. For example, if the subrole DVA called employee-status of the Person class is REQUIRED, an entity cannot participate only in the Person class. It must participate in one of the subclasses of Person, namely, Employee or Previous-Employee.

#### READONLY

The READONLY option prevents the DVA from being updated. A READONLY DVA cannot be assigned a value by means of an OML *INSERT* or *MODIFY* statement. READONLY is especially useful when applied in a schema change to a DVA that has been set to final values. For example, in an application involving taxes, a tax-rate DVA could be set to READONLY after tax rates have been established to prevent existing values from being modified.

All subrole DVAs are automatically READONLY since their value is maintained by the system. By default, all other DVAs are not READONLY.

### SV or SINGLEVALUED

By default, all DVAs are single valued (SV). SV, or its synonym SINGLEVALUED, can be explicitly stated for documentation purposes. A single-valued DVA can only be given one value for each entity in the owning class.

When a subrole DVA is single valued, the subclasses of the class that owns the DVA are considered to be mutually exclusive. That is, since the DVA can only have one value, an entity can only participate in one subclass at the next level below the class that owns the DVA. For example, the subrole DVA employee-status of the Person class is single valued; therefore, a Person can be an Employee or a Previous-Employee, but not both.

### MV or MULTIVALUED

MV and MULTIVALUED are synonyms. The MV option allows a DVA to be given multiple values for each entity in the owning class.

When a subrole DVA is multivalued, the subclasses of the class that owns the DVA are considered to be nonexclusive. That is, since the DVA can have multiple values, an entity can participate in any number of subclasses at the next level below the class that owns the DVA. For example, the subrole DVA employee-roles of the Employee class is multivalued; therefore, an Employee can be a Project-Employee, a Manager, or both.

### DISTINCT

The DISTINCT option is only allowed for DVAs that are multivalued. This option ensures that no two values are the same for a given entity. Note the difference between the options UNIQUE and DISTINCT: UNIQUE assures uniqueness among DVA values used in a class, while DISTINCT assures uniqueness among DVA values used for each entity. UNIQUE is a stronger constraint than DISTINCT, and specifying DISTINCT for a UNIQUE DVA is inconsequential.

Multivalued subrole DVAs are always DISTINCT. This means that an entity can participate in each subclass role only once. By default, all other multivalued DVAs are not DISTINCT.

### MAX <limit>

The MAX <limit> option is only allowed for DVAs that are multivalued. This option limits the number of values that the multivalued DVA can have for each entity. The option must specify a limit that is a positive integer greater than 0. If the DVA is not REQUIRED, the number of values that it can have for each entity must be between 0 and the specified limit. If the DVA is REQUIRED, the number of values that it can have for each entity must be between 1 and the specified limit.

By default, multivalued nonsubrole DVAs have no MAX limit, thereby allowing an arbitrary number of values. Multivalued subrole DVAs have an implicit MAX limit equal to the total number of subclasses that you define as subroles of the class. You can specify an explicit MAX limit for multivalued subrole DVAs, but the limit cannot be greater than the implicit MAX limit.

**INITIALVALUE**

The INITIALVALUE option allows a default value to be given for a DVA when an entity is inserted into the database without assigning an explicit value for the DVA. If a DVA is defined without the INITIALVALUE option, then insertion of an entity without assigning an explicit value for the DVA results in the DVA being null for that entity. If the DVA is defined with the INITIALVALUE option, the entity inserted without an explicit value for the DVA assumes the value specified in the <initial value> construct of the INITIALVALUE option.

The INITIALVALUE option is not allowed for compound or subrole DVAs, and it is not allowed for multivalued DVAs. The specified initial value must be compatible with the type of the DVA. For example, a <string> constant must be used when the DVA is a string, the value TRUE or FALSE must be used when the DVA is Boolean, and so on.

The INITIALVALUE option is especially useful for meeting the REQUIRED constraint when entities are inserted into the database. However, the INITIALVALUE option is impractical for a UNIQUE DVA because the option causes SIM to attempt to assign the same value to the DVA for each entity.

**Example**

The following example shows four DVAs in the base class declaration for the Person base class. (For the complete declaration, refer to “ORGANIZATION Database” in Section 3, “Basic ODL Considerations.”)

```

CLASS Person      "Persons related to the company"
  (employee-status "Whether person is a current or previous "
    "employee of the company"
    : SUBROLE (Employee, Previous-Employee);
  person-id       "Unique person identification"
    : INTEGER (1..99999), UNIQUE, REQUIRED;
  marital-status  "Current marital status"
    : SYMBOLIC (single, married, divorced),
      REQUIRED, INITIALVALUE single;
  children        "Person's children (optional)"
    : STRING [20], MV (DISTINCT, MAX 10);
);

```

Employee-status is a subrole DVA that identifies two subclasses for the Person base class: Employee and Previous-Employee. Since employee-status is single valued by default, a Person can participate in only one of these subclasses. However, since employee-status is not of the type REQUIRED, an entity can be a Person only without having to participate in either Employee or Previous-Employee.

Person-id is a single-valued DVA that must be an integer between 1 and 99999. Since person-id is UNIQUE and REQUIRED, every Person entity must have a unique person-id.

Marital-status is a DVA of the type REQUIRED. If no value is given for marital-status when a Person entity is inserted, the value SINGLE is assigned by default.

Children is a multivalued DVA that can hold the names of up to 10 children for each Person entity. The children name values must be distinct from one another (no duplicates) for each Person entity.

### Entity-Valued Attributes (EVAs)

Information on defining entity-valued attributes (EVAs) in a base class declaration or a subclass declaration is presented in the following paragraphs.

An EVA is an attribute that establishes relationships between entities. In contrast to the values of a DVA, the values of an EVA are not printable. Rather, these values are relationship instances that reference entities within the target class for the EVA. The target class is the base class or the subclass to which the EVA points. Hence, each EVA has two classes of interest:

- Its owning or perspective class, that is, the class in which it is defined
- Its target class, to which it points

With SIM, entity relationships are bidirectional. A relationship is defined by a pair of EVAs that are inverses of each other. EVA defines the relationship from the perspective of its owning class. You can specify attribute options for an EVA to define an inverse EVA, a relationship type, or integrity constraints. For example, by specifying both EVAs of a relationship as single valued (SV), you define a relationship type that is one-to-one. Table 5–1 shows how various relationship types are constructed from EVAs that are explicitly declared as inverses of each other.

**Table 5–1. Relationships between Explicitly Declared EVAs**

| EVA Options | INVERSE Options | Relationship Type   |
|-------------|-----------------|---|
| SV          | SV              | One-to-one relationship (both EVAs are automatically UNIQUE)                          |
| MV          | SV              | One-to-many relationship (multivalued EVA is automatically UNIQUE)                    |
| SV          | MV              | Many-to-one relationship (multivalued EVA is automatically UNIQUE)                    |
| MV          | MV              | Many-to-many nondistinct relationship (duplicate relationship instances are allowed)  |
| MV DISTINCT | MV DISTINCT     | Many-to-many distinct relationship (duplicate relationship instances are not allowed) |

Note that when an EVA is declared as SV, its INVERSE is automatically declared as UNIQUE.

If you declare only one EVA of a relationship, SIM automatically declares an implicit, unnamed inverse EVA. This implicit EVA assumes the most general options possible (namely, those options that afford the least restrictions), as shown in Table 5–2.

Table 5–2. Relationships between Declared EVA and Implicit EVA

| EVA Options | Implicit INVERSE Options | Relationship Type                     |
|-------------|--------------------------|---------------------------------------|
| SV UNIQUE   | SV UNIQUE                | One-to-one relationship               |
| SV          | MV UNIQUE                | Many-to-one relationship              |
| MV UNIQUE   | SV                       | One-to-many relationship              |
| MV          | MV                       | Many-to-many nondistinct relationship |
| MV DISTINCT | MV DISTINCT              | Many-to-many distinct relationship    |

Table 5–2 shows that an inverse EVA does not have to be defined to create a particular kind of relationship. However, explicitly declaring both EVAs has the following two advantages:

- It enables you to add more integrity constraints, such as the **REQUIRED** or **MAX** options, to each EVA.
- It enables you to update the relationship from the perspective of either EVA. When one EVA is implicitly declared, you cannot update the relationship from the perspective of the implicit EVA.

Relationships are called binary because each relationship instance involves two entities. A relationship in which both EVAs are in the same hierarchy is called reflexive. With a reflexive relationship, a single entity can assume the role at both ends of a relationship instance; for example, a person can be his or her own tax adviser. SIM also allows an EVA to be its own inverse EVA. For example, the inverse of spouse is spouse. This construct is called a self-reflexive EVA.

SIM provides full referential integrity for relationships. You are not allowed to update an EVA in such a way that it points to an invalid entity. Also, whenever you update an EVA, the inverse EVA is automatically updated to reflect the new or deleted relationship instances. Whether an EVA is updated explicitly or implicitly, SIM ensures that the update complies with all applicable integrity constraints.

### Syntax

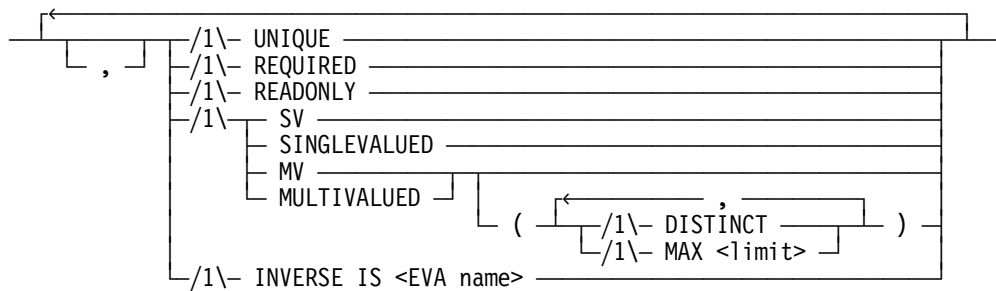
The following diagrams show the syntax to use for defining an entity-valued attribute (EVA):

#### <entity-valued attribute>

```

—<EVA name> ————— : —<base class name> —————
                        |<comment>|                     |<subclass name>| |<EVA options>|

```

**<EVA options>****Explanation**

The syntax elements shown in the diagrams are explained in the following table and in the paragraphs that succeed the table.

| Syntax Element    | Explanation   |
|-------------------|---|
| <EVA name>        | Specifies the name of the EVA that you are defining. The name must be a valid SIM identifier, namely, any combination of letters, digits, hyphens, or underscores, starting with a letter and ending with a letter or digit, up to 30 characters in length. |
| <comment>         | Denotes optional remarks about the EVA, such as a brief description of its purpose. This comment is to be included in the schema file and saved in the SIM database directory. The comment must be enclosed in quotation marks (" ").                       |
| <base class name> | Identifies a base class to which the EVA points.  |
| <subclass name>   | Identifies a subclass to which the EVA points.  |
| <EVA options>     | Indicates characteristics of the EVA, such as integrity constraints. The EVA options are described the paragraphs that follow this table.   |

**EVA Options**

You optionally can specify EVA options for an EVA to identify an inverse EVA, a relationship type, or integrity constraints. When an EVA option serves as a constraint, any OML statement that attempts to violate the constraint is aborted and an error message results. If you add a constraint to an existing EVA by means of a CHANGE command, SIM ensures that all existing data meets the new constraint. (Refer to Section 12, "Changing a SIM Database Schema," for restrictions and implications with respect to changing a schema.)

Each EVA option is explained in the following paragraphs.

**UNIQUE**

The **UNIQUE** option ensures that each target entity referenced by the EVA is only referenced once. The option has the same effect as declaring the inverse EVA as single valued (SV). A multivalued EVA cannot have an inverse EVA that is **UNIQUE**; any attempt to establish this relationship results in an error. In general, you do not need to specify the **UNIQUE** option for EVAs. If you simply specify the correct cardinality for each EVA in a relationship (SV or MV), SIM automatically determines when the EVA is **UNIQUE**.

**REQUIRED**

The **REQUIRED** option ensures that the EVA is never null. When applied to a multivalued EVA, **REQUIRED** ensures that the EVA has at least one value. By default, EVAs are not **REQUIRED**.

Note that you cannot insert both entities in a relationship instance in a single OML statement. Therefore, SIM does not allow you to declare both EVAs of a relationship as **REQUIRED** when you use the **ADD** command to create a schema since you would be unable to populate the relationship. However, you can include the **REQUIRED** option in one or both EVAs of a relationship when you use the **CHANGE** command to change a schema provided that all existing entities satisfy the **REQUIRED** constraint.

**READONLY**

The **READONLY** option prevents the EVA from being updated. A **READONLY** EVA cannot be assigned a value explicitly or implicitly by means of an OML *INSERT* or *MODIFY* statement. **READONLY** is especially useful when applied in a schema change to an EVA that has been set to final values. By default, EVAs are not **READONLY**.

**SV or SINGLEVALUED**

By default, all explicitly declared EVAs are single valued (SV). A single-valued EVA can only reference one target entity for each entity in the class that owns the EVA. **SV**, or its synonym **SINGLEVALUED**, can be explicitly stated for documentation purposes. The cardinality of an EVA and the cardinality of its inverse EVA together define the type of the relationship they form.

**MV or MULTIVALUED**

**MV** and **MULTIVALUED** are synonyms. An EVA can be multivalued only when its inverse EVA is not **UNIQUE**. A multivalued EVA can have multiple values (relationship instances) for each entity that participates in the class that owns the EVA.

**DISTINCT**

The **DISTINCT** option is only allowed for EVAs that are multivalued. This option ensures that the EVA does not reference the same target entity twice for any entity of the perspective class. Note that a multivalued EVA is automatically **UNIQUE** if its inverse EVA is single valued; this is a stronger constraint than **DISTINCT**. Therefore, specifying **DISTINCT** for a multivalued **UNIQUE** EVA is inconsequential.

DISTINCT is only meaningful when you specify it for a multivalued EVA that has an inverse EVA that is also multivalued. The relationship type formed by such an EVA pair is many-to-many, and by default SIM allows such relationship types to have duplicate instances. If you specify DISTINCT for one EVA, DISTINCT is also assumed by its inverse EVA. Thus when you specify DISTINCT for either of the two EVAs, duplicate instances are not allowed.

Although SIM allows duplicates in many-to-many relationships by default, many-to-many relationships in applications typically should not allow duplicates. Therefore, most multivalued EVAs that have multivalued inverse EVAs should be declared as DISTINCT.

### MAX <limit>

The MAX <limit> option is only allowed for EVAs that are multivalued. This option limits the number of values that the multivalued EVA can have for each perspective entity. The option must specify a limit that is a positive integer greater than 0. If the EVA is not REQUIRED, the number of values that it can have for each entity must be between 0 and the specified limit. If the EVA is REQUIRED, the number of values that it can have for each entity must be between 1 and the specified limit.

### INVERSE IS <EVA name>

This option specifies the given EVA (the <EVA name> parameter) as the inverse of the EVA you are currently defining. The given EVA must be defined in the schema, and if that EVA specifies an inverse EVA, it must be the EVA you are currently defining. An EVA and its inverse EVA form a relationship, and attribute options of each EVA define characteristics of the relationship.

### Example

The following example shows three EVAs in the base class declaration for the Project base class. (For the complete declaration, refer to “ORGANIZATION Database” in Section 3, “Basic ODL Considerations.”)

```
CLASS Project          "Current and complete Projects"
  (project-manager      "Current project manager"
    : Manager, INVERSE IS projects-managing;
    project-members     "Current employees on project"
      : Project-Employee, MV (DISTINCT, MAX 20),
      INVERSE IS current-projects;
    dept-assigned       "Responsible department"
      : Department, REQUIRED;
  );
```

Project-manager is a single-valued EVA that points to the Manager subclass; its inverse EVA is projects-managing. Assuming that projects-managing is defined as multivalued, then project-manager and projects-managing form a many-to-one relationship.



Project-members is a multivalued EVA that points to the Project-Employee subclass, and its inverse EVA is current-projects. Assuming that current-projects is defined as multivalued, then project-members and current-projects form a many-to-many relationship in which duplicate instances are not allowed. Moreover, a Project entity cannot have more than 20 project-members.

Dept-assigned is a single-valued EVA with no inverse EVA specified. Therefore, its inverse EVA is implicitly declared as multivalued, thereby forming a many-to-one relationship.



# Section 6

## Indexes

This section describes the use of ODL to define indexes and presents information, including restrictions, on using indexes.

All examples presented in this section reflect the ORGANIZATION database, which is described in Section 3, “Basic ODL Considerations.”

### Purposes for Declaring Indexes

An index is a logical SIM construct that you can use to optimize query performance, enforce uniqueness constraints, or provide a basis for specifying DMSII mapping options.

Each index has three parts:

- Name
- Target
- Key

The name of an index is a SIM identifier. This name is never referenced in the SIM Object Manipulation Language (OML). It can only be referenced in other parts of the database schema.

The target of an index is a base class or a subclass that the index spans. The index applies only to entities that participate in its target class.

The key of an index consists of one or more data-valued attributes (DVAs) that determine the way in which the index is to be used.

You can declare an index for any or all of the following three basic purposes:

- To provide an auxiliary path that can be used to access entities. Such an index allows entities to be efficiently retrieved based on the value of its key. When a query is performed, the SIM query optimizer automatically chooses any indexes that can be used to increase the speed of the query.
- To enforce a uniqueness constraint on the key values of the index. An index can be declared to ensure that every data value represented by its key exists only once.
- To give a name to the otherwise unnamed index that is automatically created on behalf of a UNIQUE DVA. Naming the index enables you to control the physical

mapping of the index by referencing it in a DMSII mapping-options declaration. In fact, DMSII mapping options can be specified for all user-declared indexes.

Refer to Section 9, “DMSII Mapping Options,” for more information on mapping options for indexes.

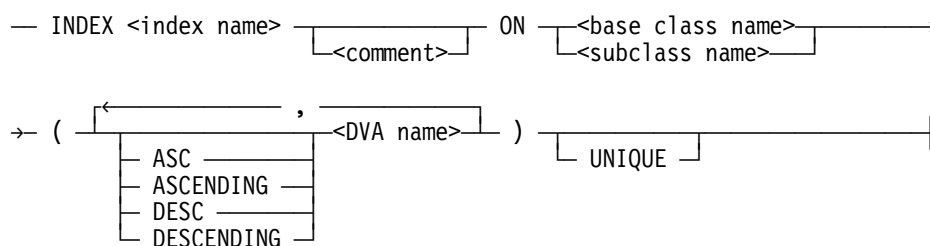
## Index Declaration

Information on defining an index through an index declaration is presented in the following paragraphs.

## Syntax

The following diagram shows the syntax to use for defining an index:

## &lt;index declaration&gt;



### Explanation

The syntax elements shown in the diagram are explained in the following table:

| Syntax Element       | Explanation   |
|----------------------|---|
| INDEX <index name>   | Identifies uniquely the index that you are defining. The index name must be a valid SIM identifier. That is, it can be any combination of letters, digits, hyphens, or underscores, except that it must start with a letter, end with a letter or digit, and be no longer than 30 characters. |
| <comment>            | Denotes optional remarks about the index, such as a brief description of its purpose. This comment is to be included in the schema file and saved in the SIM database directory. The comment must be enclosed in quotation marks (" ").   |
| ON <base class name> | Identifies a base class as the target class for the index.  |
| ON <subclass name>   | Identifies a subclass as the target class for the index.  |
| ASC                  | Specifies that the following DVA is to serve as an ascending key. Ascending is the default ordering option. With an ascending key, data are ordered starting with the lowest value of the key and ranging to the highest value of the key.  |

| Syntax Element | Explanation  |
|----------------|--|
| ASCENDING      | Same as ASC.   |
| DESC           | Specifies that the following DVA is to serve as a descending key. With a descending key, data are ordered starting with the highest value of the key and ranging to the lowest value of the key. |
| DESCENDING     | Same as DESC.  |
| <DVA name>     | Identifies a DVA that is to serve as an index key.   |
| UNIQUE         | Ensures that each value corresponding to the index key specification is never used twice.  |

Note that you can specify more than one index declaration on the same class.

### Example

The following example shows an index declaration on the Person base class:

```
INDEX Person-by-id ON Person
(person-id);
```

Since person-id is declared as UNIQUE within the Person class, SIM automatically generates an unnamed index to enforce the uniqueness constraint. Therefore, the index declaration shown above merely names this index, which enables it to be referenced in a DMSII mapping-options declaration. Besides enforcing the uniqueness constraint, this index can be used to efficiently retrieve Person entities by means of specific person-id values.

## Restrictions on All Indexes

For all indexes, the key of an index must observe the following restrictions:

- Each specified DVA name must correspond to an immediate DVA of the target class of the index. A key attribute cannot be inherited.
- An index key can be either a single multivalued DVA or one or more single-valued DVAs. A multivalued DVA can only be used alone as an index key.
- A key attribute cannot be a DVA defined as a subrole DVA.
- If a DVA of the type NUMBER is a key attribute of any index, it must be signed with a precision of 11 digits or less, or it must be unsigned with a precision of 12 digits or less. These restrictions are due to DMSII limitations.
- If a compound DVA is a key attribute of an index, the NUMBER restrictions indicated above apply to all of the components of the compound.

# Restrictions on UNIQUE Indexes

In addition to the restrictions that apply to all indexes, the following restrictions apply to UNIQUE indexes:

- If any of the key attributes of an index are UNIQUE, the index is automatically considered by SIM as UNIQUE. (Explicitly specifying UNIQUE in the index declaration in this situation is unnecessary but acceptable.)
- The uniqueness constraint is enforced only for entities for which all key attributes are nonnull. This restriction is necessary because the uniqueness constraint corresponding to a specific index cannot be evaluated for an entity when one of the key attributes corresponding to that index is null.
- If the first key of a UNIQUE index is a compound DVA, the uniqueness constraint is enforced only when all of the components of the DVA are nonnull. Furthermore, the index is not available for optimizing query execution because the implementation of the index prevents it from being usable by queries.
- All secondary keys (that is, the second and subsequent keys) of a UNIQUE index must be declared as REQUIRED.
- The secondary keys of a UNIQUE index cannot be DVAs of the type compound.
- Declaring a UNIQUE index with a single key attribute has the same effect as declaring the attribute itself as UNIQUE. This fact is especially significant if the key is a multivalued DVA, in which case a uniqueness constraint is applied to all nonnull values of the DVA for all entities of the target class.
- If an index key consists of a single UNIQUE DVA, the index effectively names the index that SIM automatically generates on behalf of the DVA. One advantage of naming an otherwise unnamed index is that naming the index allows you to reference it in a DMSII mapping-options declaration.

## Guidelines for Using Indexes

It is important to understand the ways in which you can use an index for optimizing query executions. In all cases but one, an index is made known to the SIM query optimizer, thereby allowing the index to be used during query optimization. Only when an index is UNIQUE and its first key attribute is a compound DVA is the index unusable by the query optimizer.

You can always use an index that is known to the query optimizer for equality searches. During an equality search, the index is used to locate entities whose key attributes exactly match a designated value (for example, <DVA name> = <value>).

For an index to be used for inequality searches (for example, <DVA name> >= <value>), it must be mapped to a sequential-capable structure. By default, SIM maps an index to a sequential-capable structure, such as an INDEX SEQUENTIAL set. You can override this mapping by specifying mapping options. (Refer to Section 9, "DMSII Mapping Options," for details on mapping options for indexes.)

When an index has multiple key attributes, the order in which the attributes are specified in the key is important. This order is important because SIM has the ability to use less than the full key of an index for searches, but it must use key attributes in consecutive order, beginning with the leftmost attribute. Furthermore, only the rightmost key attribute that you use in a query can be used for an inequality search. It can optionally be used for an equality search. All of the other key attributes that you use in the query must be used for equality searches.

### Examples

Consider the following portion of the ORGANIZATION database schema:

```
SUBCLASS Employee      "Current employees of the company"
  OF Person
  (hire-date           : DATE;
   salary              : NUMBER [8, 2];
   exemption-status    : SYMBOLIC (exempt, non-exempt);
  );

INDEX Employee-index ON Employee
  (hire-date, salary, exemption-status);
```

Both of the following queries could exploit the presence of this index:

```
RET name OF Employee
WHERE exemption-status = exempt

RET name OF Employee
WHERE exemption-status = non-exempt AND
      hire-date >= 7/27/1990
```

Both of these queries use only a part of the index key, but each query always uses attributes in order from left to right. Furthermore, the inequality used in the second query appears on the rightmost attribute used. A query that searches for employees based on salary alone could not use this index (although it might use another index).

Consider a situation in which the WHERE clause in the second query is amended to read as follows:

```
RET name OF Employee
WHERE exemption-status = non-exempt AND
      hire-date >= 7/27/1990 AND
      salary > 30000
```

This query could use the index to efficiently find entities that meet the stated exemption-status and hire-date conditions. However, the index would not be helpful for the salary condition because an inequality was specified for the previous key attribute.

An index is most efficiently used when all of its key attributes are used for equality searches, except that the last key attribute can be used for either an equality search or an inequality search.





# Section 7

## Verifies

Verifies provide a means of establishing conditions that must be satisfied by a SIM database. This section describes the use of ODL to define verifies and presents guidelines for using verify declarations.

All examples presented in this section reflect the ORGANIZATION database, which is described in Section 3, “Basic ODL Considerations.”

### When to Declare a Verify

A semantic database is intended to capture as much as is practicable of the semantics of an application. Therefore, the database should impose the maximum possible number of constraints on the data, consistent with ease of use and the scope and purpose of the application. Constraining the data has the effect of extending the range of inferences implied by the database with respect to the application. In the design of a SIM database, the constraints should be established wherever possible, first, through proper class/subclass structural relationships and, second, through attribute constraints. The use of verifies is an effective way to deal with the special cases, that is, a way to achieve the needed constraints that cannot practicably be established through class/subclass relationships and attribute constraints.

Verifies are an important feature that should be used discreetly. You should declare a verify to achieve a needed constraint only when you have determined that no other way is practicable. You should consider the impact of the verify on the application in terms of performance. In particular, a verify might significantly degrade performance if the condition involves aggregate functions or multivalued EVAs.

### Techniques for Using Verifies

A suggested technique for using verifies is to put them in effect only when they are needed. For example, during system prototyping and development—when performance is usually not an important consideration—all desired verifies can be declared in the database schema. Once application programs have been tested and stabilized, expensive verifies (that is, those that involve substantial processing) can be turned off. Turning them off means removing them from the schema (for example, by using the percent sign (%) to comment out the verifies) and then processing the schema through the CHANGE DATABASE command.

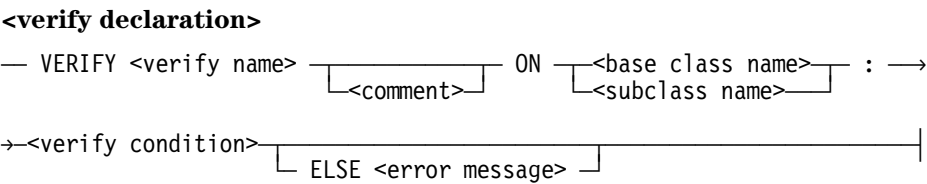
Another technique for using verifies is to turn off the verifies that significantly degrade performance before you populate a database, and then to reactivate these verifies after the database has been loaded. Adding or changing a verify through the CHANGE DATABASE command causes that verify to be checked for validity.

## Verify Declaration

Information on defining a verify through a verify declaration is presented in the following paragraphs.

### Syntax

The following diagram shows the syntax to use for defining a verify:



### Explanation

The syntax elements shown in the diagram are explained in the following table:

| Syntax Element       | Explanation   |
|----------------------|---|
| VERIFY <verify name> | Identifies uniquely the verify condition that you are defining. A verify name must be a valid SIM identifier. That is, it can be any combination of letters, digits, hyphens, or underscores, except that it must start with a letter, end with a digit, and be no longer than 30 characters.<br><br>The verify name that you specify will be associated with any error message resulting from violation of the verify condition. |
| <comment>            | Denotes optional remarks about the verify condition, such as a brief description of its purpose. This comment is to be included in the schema file and saved in the SIM database directory. The comment must be enclosed in quotation marks (" ").  |
| ON <base class name> | Identifies a base class as the perspective class for the verify.  |
| ON <subclass name>   | Identifies a subclass as the perspective class for the verify.  |

| Syntax Element       | Explanation   |
|----------------------|---|
| <verify condition>   | <p>Specifies a condition that must be satisfied by the database. The condition is an OML global selection expression that returns a Boolean result: true (T), false (F), or null (?). For information on the operators and functions that you can use to construct an OML global selection expression, refer to the <i>SIM OML Programming Guide</i>. Note, however, that you cannot use transitive functions or the operators CURRENT_DATE or CURRENT_TIME in the expression for a verify.</p> <p>For more information about the verify condition, refer to “How Verifies Are Used” and “When Verifies Are Valid” in this section.</p> |
| ELSE <error message> | <p>Defines the error message that is to be returned with the verify name when the verify condition is violated. The error message must be enclosed in quotation marks (“ ”).</p> <p>If you do not specify <i>ELSE &lt;error message&gt;</i>, the default system error message “Semantic integrity constraint violation: &lt;verify name&gt;” is returned when a violation occurs.</p> <p>The error message is returned either to the user’s terminal or to the program, depending on the interface used. An error message returned to a program can be accessed through the appropriate host language function.</p>                     |

**Note that when you declare a verify that ends with a pattern-matching expression and no ELSE clause, you must separate the pattern and the final semicolon by at least one blank space. This blank space prevents the semicolon from being interpreted as part of the pattern. The following example illustrates this point:**

```
VERIFY Valid-department-name ON Department:
  dept-name ISIN {A..Z" "-}*;    % note space before semicolon
```

# How Verifies Are Used

Any verify declaration included in a database schema specifies a condition that must be satisfied by the database. The verify condition is an OML global selection expression that returns a Boolean result: true (T), false (F), or null (?). The expression can have only one class of interest, namely, the perspective class.

The verify condition typically does one of the following:

- Defines how data in the database is to be compared with some other data that resides in the same class or a different class
- Defines how data in the database is to be compared with a constant value
- Asserts the existence of certain information in the database

The verify condition is checked whenever an action is taken to insert or delete an entity, or whenever an action is taken to modify a relevant attribute from the perspective class or a related class. The insertion, deletion, or modification is permitted by SIM only if the constraint of the expression is met. If the constraint is not met, the update statement is aborted and an appropriate error message is displayed.

A verify is triggered by the occurrence of any update that has a potential for violating the verify condition. For example, assume that a certain INSERT statement can violate a verify. The INSERT statement is processed first. Then the verify condition is tested to ensure that it is still valid for the database. If the verify condition fails the test, the statement to insert the entity is aborted.

# Locking Implications in Using Verifies

It is important to note that verifies have an impact on the amount of data that can be locked during the processing of an update query. Testing a verify often involves a single entity or only a few entities. However, in some cases, such as those in which a verify uses aggregates, quantifiers, or multivalued attributes, SIM might need to examine many entities to enforce a verify. In these cases, all entities examined are share locked, thereby preventing them from being exclusive locked by another program. Hence, certain verifies might have substantial locking implications.

## When Verifies Are Valid

Any verify is defined to be valid by SIM if and only if its verify condition is either true or null. That is, a given verify declaration is defined to be valid if the following retrieval never returns a false value:

```
FROM <base class or subclass> RETRIEVE <verify condition>
```

If the verify condition is multivalued, then the verify is valid only if none of the values for the attributes of the verify condition is false. The values must be either true or null.

A verify is checked for validity during ODL processing when it is first declared and whenever its verify condition is changed. The new or changed verify condition is applied to the database when the schema file is processed. The database schema cannot change if the database does not satisfy this verify condition. If the database does satisfy the condition, the new or amended verify declaration becomes part of the database schema provided that the schema file meets all other processing constraints.

## Restriction on Adding or Changing a Verify

A verify declaration cannot be added or changed as part of a database schema change that involves a physical database change. That is, no existing structures can be modified by adding, deleting, or modifying attributes or indexes, and no new structures can be added to the database in conjunction with an added or changed verify condition.

## Examples of Verify Declarations

The following examples demonstrate some of the ways in which verifies can be used.

### Example 1

Assume that a manager's bonus must meet two constraints:

- It must not exceed \$10,000.
- It must not exceed 10 percent of his or her annual salary.

These constraints can be enforced by means of the following verify declaration:

```
VERIFY Valid-manager-bonus ON Manager:  
  bonus LEQ 10000 AND  
  bonus LEQ salary *.1  
ELSE "A manager's bonus cannot exceed $10,000 or 10% of his or "  
     "her salary";
```

Whenever a manager's bonus or salary is updated, the two constraints specified in this verify are checked. If either constraint is violated, the corresponding insertion or modification is aborted, and the error message defined in the verify is returned to the user.

### Example 2

Assume that a constraint is specified on the database that an employee cannot earn as much as or more than his or her manager. This constraint is specified in the following verify declaration, together with an explanatory error message to be returned if the constraint is violated:

```
VERIFY Valid-employee-salary-1 ON Employee:
  salary LSS salary OF employee-manager
  ELSE "Employee's salary must be less than that of his or her "
      "manager";
```

When an entity is submitted for insertion into the Employee class of the database, the employee's salary is compared with that of his or her manager. This comparison is done by means of the employee-manager EVA of the Employee class. If the employee's salary is less than the manager's salary, the entity is inserted. If the employee's salary is equal to or more than the manager's salary, insertion of the entity is disallowed and the error message is returned.

Similarly, if the entity submitted is for an employee who is a manager, the salary of that manager is compared with that of his or her employees. This comparison is done by means of the employees-managing EVA of the Manager class, which is the inverse of the employee-manager EVA of the Employee class.

The verify is also checked for relevant entities whenever a salary, employee-manager, or employees-managing attribute is modified.

### Example 3

The following verify declaration ensures that the salary of each employee is within 20 percent of the average salary of all employees:

```
VERIFY Valid-employee-salary-2 ON Employee:
  ABS (salary - AVG (salary OF Employee)) LEQ
  AVG (salary OF Employee) *.2
  ELSE "Employee salary must be within 20% of average employee "
      "employee salary";
```

In this example, the aggregate function AVG is used twice, causing the corresponding verify to be potentially expensive to enforce. If the Employee class contains a large number of entities, this verify could impose a significant overhead to each query that triggers it.

### Example 4

Assume that for each current project in the database, either the project must not be marked as active or it must have at least two project members assigned to it. This constraint could be enforced by the following verify declaration:

```
VERIFY Valid-project-team-minimum ON Current-Project:
  NOT project-active OR COUNT (project-members) GEQ 2
  ELSE "Active projects must have at least 2 project-members";
```

# Section 8

## Security

This section describes the use of ODL to define accesses and permissions, and explains the purpose of accesses and permissions.

All examples presented in this section reflect the ORGANIZATION database, which is described in Section 3, “Basic ODL Considerations.”

Access and permission declarations, when used together, provide a means to establish security constraints on a SIM database. An access defines a window into a set of data, and a permission grants an access to users and programs.

When a SIM database schema is defined without any accesses or permissions, the database is considered public. Once a program or user successfully opens a public database, SIM does not restrict the program or user from accessing the database for security reasons. When at least one access is defined in a database schema, the database is considered controlled. When a program or user opens a controlled database, that program or user can access the database only as explicitly allowed by the accesses and permissions for the database.

Accesses and permissions provide logical database security. They only control a program's accesses to a SIM database after that program opens the database. The ability to open a SIM database or to access database files outside of the database system (for example, by using normal file I/O) is considered physical database security. Physical database security of a SIM database is controlled by DMSII mapping-option declarations that specify constructs such as guard files. DMSII mapping-option declarations are described in Section 9, “DMSII Mapping Options.”

## Accesses

An access essentially defines a window into a set of data within a SIM database and into a set of data manipulation operations (OML operations) that are allowed on the data within that window. If an access is declared in a database schema, the access declaration must precede any permission declarations with which it is associated.


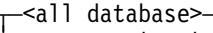
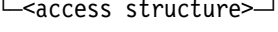
## Access Declaration

Information on defining an access through an access declaration is presented in the following paragraphs.

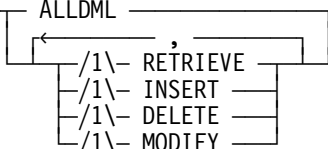
### Syntax

The following diagrams show the syntax to use for defining an access:


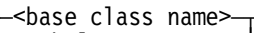
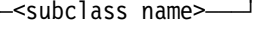
#### <access declaration>

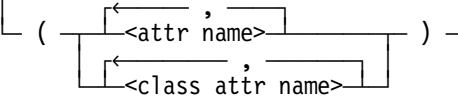
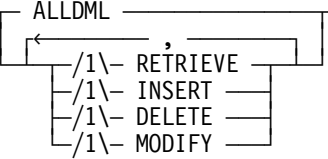
— ACCESS <access name>  ON  


#### <all database>

— ALLDB 

#### <access structure>

→  

→ 



**Explanation**

The syntax elements shown in the diagrams are explained in the following table:

| Syntax Element        | Explanation   |
|-----------------------|---|
| ACCESS <access name>  | Identifies uniquely the access that you are defining. The access name must be a valid SIM identifier. That is, it can be any combination of letters, digits, hyphens, or underscores, except that it must start with a letter, end with a letter or digit, and be no longer than 30 characters.                                 |
| <comment>             | Denotes optional remarks about the access, such as a brief description of its purpose. This comment is to be included in the schema file and saved in the SIM database directory. The comment must be enclosed in quotation marks (" ").  |
| ON <all database>     | Specifies that the allowed OML operations apply to the entire database. Refer to "ALLDB Access," which follows this table, for more information.  |
| ON <access structure> | Specifies that the allowed OML operations apply to the database as defined in the <access structure> construct. There are two forms of access structures: class attributes and attributes. Refer to "Class Attribute Access" and "Attribute Access," which follow this table, for more information.                             |
| ALLDML                | Specifies that RETRIEVE, INSERT, DELETE, and MODIFY operations are allowed. For descriptions of these OML operations, refer to the explanations of the syntax elements RETRIEVE, INSERT, DELETE, and MODIFY that follow. (For detailed information about these OML operations, refer to the <i>SIM OML Programming Guide</i> .) |
| RETRIEVE              | Specifies that RETRIEVE operations are allowed. RETRIEVE operations are used to inquire on entities of a base class or subclass.  |
| INSERT                | Specifies that INSERT operations are allowed. INSERT operations are used to add entities to a base class or subclass.   |
| DELETE                | Specifies that DELETE operations are allowed. DELETE operations are used to delete entities from a base class or subclass.  |
| MODIFY                | Specifies that MODIFY operations are allowed. MODIFY operations are used to modify entities of a base class or subclass.  |

| Syntax Element                   | Explanation   |
|----------------------------------|---|
| CLASS-ATTRIBUTES OF              | Specifies that the access constraints apply only to class attributes of the designated base class or subclass.  |
| <base class name>                | Specifies the name of the base class to which the access constraints apply.   |
| <subclass name>                  | Specifies the name of the subclass to which the access constraints apply.   |
| <attr name>                      | <p>Specifies the name of an attribute of the designated class to which access is allowed. For example, if you specify the name attribute of the Person class, access to the names of persons is allowed in an OML query.</p> <p>Note that if you specify the names of one or more attributes of the designated class, access is allowed only to the specified attributes. If you do not specify any attribute names, access is allowed to all attributes of the class.</p>  |
| <class attr name>                | <p>Specifies the name of a class attribute of the designated class to which access is allowed.</p> <p>Note that if you specify the names of one or more class attributes of the designated class, access is allowed only to the specified class attributes. If you do not specify any class attribute names, access is allowed to all class attributes of the class.</p>  |
| WHERE <OML selection expression> | <p>Defines a restriction that limits an OML query to a portion of a base class or subclass. Any restriction that applies to a base class also applies to its subclasses.</p> <p>The &lt;OML selection expression&gt; construct is a global expression that can involve only immediate, nonsubrole, single-valued DVAs. OML aggregates, quantifiers, and transitive functions such as AVERAGE and SOME are not allowed.</p> <p>For information on the operators and functions that you can use to construct an OML global selection expression, refer to the <i>SIM OML Programming Guide</i>.</p> |

There are three basic types of access declarations, each of which controls security of a specific type of data. These basic access types are the following:

- ALLDB access
- Class attribute access
- Attribute access

Each of these basic access types and its usage are described in the following paragraphs.

## **ALLDB Access**

You define an ALLDB access by using the following form in an access declaration:

```
ACCESS <access name> ON ALLDB ...
```

An ALLDB access creates a window to the entire database, and it allows one or more OML verbs on data contained in that window. As shown in the syntax diagrams for an access declaration, an ALLDB access does not define any class, subclass, or attribute names, and it cannot specify a WHERE condition.

An ALLDB access must specify which OML verbs are allowed to act on the database by means of that access. It can specify ALLDML, to indicate that all OML verbs are allowed, or it can name explicitly the OML verbs that are allowed. Each program that is granted an ALLDB access by means of a permission declaration can access the entire database with the verbs specified in the access.

### Example

The following example defines an ALLDB access called Database-modify-access, which creates a window to the entire database and allows the OML verbs RETRIEVE and MODIFY:

```
ACCESS Database-modify-access "Modify access to entire database"
ON ALLDB
RETRIEVE, MODIFY;
```

All programs that are granted this access by means of a permission declaration can retrieve and modify any data in the database. Such programs cannot insert or delete entities, however, unless some other access or accesses grant these capabilities.

## Class Attribute Access

You define a class attribute access by using the following form in an access declaration:

```
ACCESS <access name> ON CLASS-ATTRIBUTES OF
<base class or subclass name> ...
```

This type of access creates a window to one or more class attributes of the specified base class or subclass. If no class attribute is specified, the window includes all class attributes of the given class. Otherwise, the window includes only the specified class attributes. A class attribute access cannot specify a WHERE condition. Multiple class attribute accesses can be declared on the same class.

As is the case with ALLDB accesses, a class attribute access must specify which OML verbs are allowed to act on the database by means of that access. It can specify ALLDML, or it can specify individual OML verbs. Note, however, that only the OML verbs RETRIEVE and MODIFY are meaningful for class attributes since class attributes are never updated by INSERT or DELETE queries. Each program that is granted a class attribute access by means of a permission declaration can access the corresponding class attributes with the verbs specified in the access.

### Example

The following example declares a class attribute access called Employee-class-att-access that provides RETRIEVE access to two class attributes of the Employee class:

```
ACCESS Employee-class-att-access "Public Employee class attributes"
ON CLASS-ATTRIBUTES OF Employee
(next-employee-id, last-hire-date)
RETRIEVE;
```

All programs that are granted this access by means of a permission declaration can retrieve the class attributes next-employee-id and last-hire-date. However, such programs cannot modify these class attributes, nor can such programs retrieve or modify the maximum-salary class attribute unless some other access or accesses grant these capabilities.

## Attribute Access

You define an attribute access by using the following form in an access declaration:

```
ACCESS <access name> ON <base class or subclass name> ...
```

This type of access creates a window to one or more attributes of the specified base class or subclass. If no attribute is specified, the window includes all immediate (but not inherited) attributes of the given class. Otherwise, the window includes only the specified attributes, which must be immediate (not inherited) attributes of the access's class. Multiple attribute accesses can be declared on the same class.

As is the case with ALLDB accesses and class attribute accesses, an attribute access must specify which OML verbs are allowed to act on the database by means of that access. It can specify ALLDML, or it can specify individual OML verbs. The OML verbs RETRIEVE, MODIFY, and DELETE provide access to existing entities, while the INSERT verb allows new entities to be inserted. If the RETRIEVE verb is allowed, the included attributes can be retrieved. If the INSERT verb is allowed, the included attributes can be assigned a value when an entity is inserted. If the MODIFY verb is allowed, the included attributes can be assigned a value when an entity is modified. If the DELETE verb is included, existing entities can be deleted regardless of which attributes are included.

An attribute access can optionally specify a WHERE condition. The corresponding OML selection expression must be restricted as indicated in the syntax element table. The WHERE condition limits the entities visible in the access's window to those that satisfy the OML selection expression. If a WHERE condition is not specified, the window includes all entities in the class of the access.

When a WHERE condition is specified for an attribute access, it only applies to the OML verbs RETRIEVE, MODIFY, and DELETE, since only these verbs affect existing entities. Therefore, if you declare an attribute access that only allows the INSERT verb, you cannot specify a WHERE condition.

## Examples of Access Declarations

### Example 1

In the following example, assume that Employee is a base class. The example declares an attribute access named Employee-update-access that allows all OML verbs to some attributes of some entities of the Employee class.

```
ACCESS Employee-update-access "Restricted Employee update access"
  ON Employee
  (employee-id, employee-manager)
  ALLDML
  WHERE exemption-status EQL non-exempt OR salary LSS 25000;
```

A program that is granted this access by means of a permission declaration can perform the following OML operations:

- The attributes employee-id, and employee-manager can be retrieved by means of a RETRIEVE query for any Employee entity that is non-exempt or that has a salary of less than 25000.
- A new Employee entity can be inserted, and the attributes employee-id and employee-manager can be assigned a value during the insertion.
- The attributes employee-id and employee-manager can be assigned a new value by means of a MODIFY query for any Employee entity that is non-exempt or that has a salary of less than 25000.
- An Employee entity that is non-exempt or that has a salary of less than 25000 can be deleted.

Other attributes of the Employee class cannot be retrieved or modified by means of this access because they are not included in the declaration of this access.

### Example 2

To perform an OML operation on an entity, a user or program must have sufficient access to all roles in which the entity participates. For example, assume that Employee in the previous example is actually a subclass of the Person base class, and that the following access is declared on Person:

```
ACCESS Person-retrieve-access    "Limited Person retrieve access"
  ON Person
  RETRIEVE
  WHERE person-id GTR 1000;
```

If it is assumed that a suitable permission grants both Person-access and Employee-access to a program, the OML operations that can be performed by the program are the following:

- All attributes of “visible Persons” (namely, Person entities with a person-id greater than 1000) can be retrieved by means of a RETRIEVE query.
- All attributes of Person and the Employee attributes employee-id and employee-manager can be retrieved by means of a RETRIEVE query for “visible Employees” (namely, Employee entities that are visible Persons and that are either non-exempt or have a salary of less than 25000).
- A visible Person that is not an Employee can be inserted into the Employee subclass. During the INSERT/FROM operation, the Employee attributes employee-id and employee-manager can be assigned a value.
- Visible Employees can be deleted from the Employee subclass (but not from the Person base class).
- The attributes employee-id and employee-manager can be assigned a new value by means of a MODIFY query for visible Employee entities.

As suggested by this list, new Person and Employee entities cannot be inserted since the INSERT verb is not allowed on Person. Similarly, Person attributes cannot be modified and Person entities cannot be deleted since neither MODIFY nor DELETE are allowed on Person.

All OML verbs are allowed on Employee. Therefore, entities can be inserted into and deleted from Employee. Moreover, designated Employee attributes can be modified as long as retrieve access is granted to the corresponding entities for both the Person and Employee classes.

## Permissions

A permission serves two purposes. First, it defines a set of subjects that match specified access constraints. Second, it grants one or more accesses to all of its subjects, thereby giving those subjects all capabilities defined by the granted accesses. In effect, a permission is an intersection between subjects and accesses.

The set of subjects defined by the permission consists of the usercodes, program names, and/or accesscodes specified in the permission. The run-time task attributes of a program that is a subject must match the specifications in the permission.

Any accesses named in a permission declaration must be defined in the schema prior to the permission declaration. A permission can name multiple accesses even when two or more accesses are of the same type and based on the same class. Moreover, an access can be named in multiple permissions.

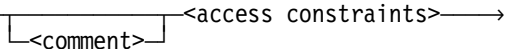
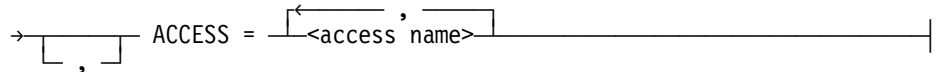
## Permission Declaration

Information on defining a permission through a permission declaration is presented in the following paragraphs.

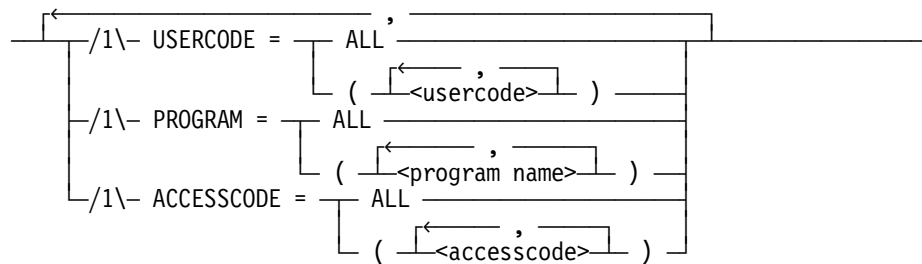
### Syntax

The following diagrams show the syntax to use for specifying a permission declaration:

#### <permission declaration>

— PERMISSION <permission name>  

#### <access constraints>



### Explanation

The syntax elements shown in the diagrams are explained in the following table:

| Syntax Element               | Explanation   |
|------------------------------|---|
| PERMISSION <permission name> | Identifies uniquely the name of the permission that you are defining. A permission name must be a valid SIM identifier, namely, any combination of letters, digits, hyphens, or underscores, starting with a letter and ending with a letter or digit, up to 30 characters in length. |
| <comment>                    | Denotes optional remarks about the permission, such as a brief description of its purpose. This comment is to be included in the schema file and saved in the SIM database directory. The comment must be enclosed in quotation marks (" ").  |
| <access constraints>         | Identifies the usercodes, programs, and accesscodes that are associated with the permission.  |
| ACCESS = <access name>       | Identifies an access associated with the permission you are defining.   |



| Syntax Element | Explanation  |
|----------------|--|
| USERCODE =     | Specifies that one or more usercodes are associated with the access.   |
| PROGRAM =      | Specifies that one or more programs are associated with the access.  |
| ACCESSCODE =   | Specifies that one or more accesscodes are associated with the access.   |
| ALL            | Specifies that all usercodes, programs, and/or accesscodes are associated with the access.   |
| <usercode>     | <p>Identifies a usercode for association with the access. An asterisk (*) can be designated, which denotes a process that is executing without a usercode.</p> <p>Note that if you specify one or more usercodes, only the specified usercodes are associated with the access.</p> |
| <program name> | <p>Identifies a program for association with the access.</p> <p>Note that if you specify one or more program names, only the specified programs are associated with the access.</p>  |
| <accesscode>   | <p>Identifies an accesscode for association with the access.</p> <p>Note that if you specify one or more accesscodes, only the specified accesscodes are associated with the access.</p>   |

As shown in the syntax diagrams, a permission must define at least one access constraint, which is a USERCODE, PROGRAM, or ACCESSCODE specification. If a constraint type is not named in a permission declaration, the specification is assumed to be ALL for that constraint type. For example, not naming usercodes in the permission declaration has the same effect as USERCODE = ALL.

Note that elements of a particular access constraint type are disjuncted (ORed), but different access constraint types are conjuncted (ANDed). This concept is shown in Example 1 under “Examples of Permission Declarations.”

### Examples of Permission Declarations

#### Example 1

Consider the following permission declaration:

```
PERMISSION Database-Production
  USERCODE = (PROD, TESTPROD),
  PROGRAM   = (*OBJECT/DBINQUIRY ON PACK,
               *OBJECT/DBUPDATE ON PACK),
  ACCESSCODE = (PAYROLL, ACCOUNTING)
  ACCESS     = Person-retrieve-access, Employee-update-access;
```

For a program to be a subject of this permission, it must satisfy the following condition:

```
(T.USERCODE = "PROD" OR T.USERCODE = "TESTPROD") AND
(T.NAME     = "*OBJECT/DBINQUIRY ON PACK" OR
 T.NAME     = "*OBJECT/DBUPDATE ON PACK") AND
(T.ACCESSCODE = "PAYROLL" OR T.ACCESSCODE = "ACCOUNTING")
```

In this example, T represents the task of the program. Consequently, T.USERCODE, T.NAME, and T.ACCESSCODE represent task attributes of the program. Note that SIM uses the task attributes of the running stack (not any intermediate libraries) when considering the security of a program.

When a program executes an OML query against a SIM database, its task attributes as of the time the database was opened are used to find permissions for which the program is a subject. The operations performed by the query are then compared to security constraints of applicable accesses. If some aspect of the query is not allowed, the query is rejected with an error.

#### Example 2

The following example of a permission declaration grants the Database-modify-access access to any program that is called \*OBJECT/DBMAINT ON PACK and that is running under the usercode PROD.

```
PERMISSION Database-Production
  USERCODE = (PROD),
  PROGRAM   = (*OBJECT/DBMAINT ON PACK),
  ACCESS     = Database-modify-access;
```

In this example, no accesscodes are specified. Therefore, accesscodes are not used to consider a program as a subject of this permission.

# Section 9

## DMSII Mapping Options

This section describes the use of ODL to declare DMSII mapping options and presents information, including guidelines, on using DMSII mapping-option declarations.

### Declaring DMSII Mapping Options

The logical description (schema) of a SIM database is mapped into a physical description (DASDL file) that produces a corresponding DMSII database. SIM uses a set of rules for automatically mapping logical constructs (such as classes, attributes, and indexes) into physical DMSII constructs (such as data sets, sets, and items). These rules reflect a balance of the following objectives, which are listed in order of priority:

- Maximum flexibility for schema changes
- Low number of DMSII structures
- Efficient use of disk space
- Reasonable performance

These rules afford a reasonable default implementation for most databases. However, they seldom afford the best possible implementation for a particular database. For example, SIM by default uses only data sets that are standard and sets that are INDEX SEQUENTIAL. Also, SIM allows DMSII to choose the default values for such parameters as ALLOWEDCORE, TABLESIZE, and POPULATION.

The default DMSII mappings are usually reasonable during logical database design, at which time the schema typically is undergoing frequent changes. These default mappings allow the designer to concentrate on achieving a correct logical design without degrading it with physical performance considerations. However, when optimum performance becomes an important consideration, the default mappings might prove to be inadequate.

Therefore, SIM allows you to declare DMSII mapping options that optimize the physical characteristics of a database. It is strongly recommended that you use these mapping options when you need to achieve such objectives as the following:

- Optimum performance
- Optimum use of disk space
- Optimum use of available memory
- Increased default limits

In some cases, especially those involving large databases, the use of DMSII mapping options might be necessary to achieving proper database operation. For example, the default POPULATION value for data sets might be insufficient for many applications.

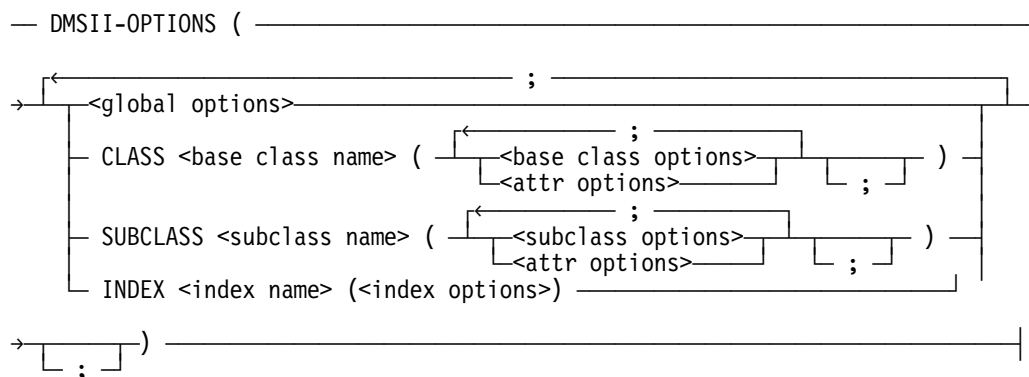
To override default mapping values, you declare the desired DMSII mapping options using ODL syntax when you create or amend the SIM database schema. If you include a DMSII mapping-options declaration in the schema file, the declaration must follow any logical constructs that it references. For example, global options can be specified anywhere in the schema, but mapping options on a class must follow the declaration of the class.

DMSII mapping options should be used in conjunction with standard DMSII monitoring and analysis tools, such as database STATISTICS, DBANALYZER, and DMMONITOR. These tools enable you to examine each physical change to determine if it achieves the desired effect. For information about these tools, refer to the *DMSII Utilities Operations Guide*.

## Syntax for Declaring Mapping Options

The following diagram shows the syntax to use for declaring DMSII mapping options:

### <DMSII mapping-options declaration>



The DMSII mapping options shown in the diagram are described in this section under the following headings:

- Global Mapping Options
- Base Class Mapping Options
- Attribute Mapping Options
- Subclass Mapping Options
- Index Mapping Options

## Global Mapping Options

Global mapping options are those options that affect the database as a whole. They include general specifications such as memory usage and auditing characteristics. They also include default specifications that are inherited by individual database components such as data sets or sets. Most of the global options provided by DMSII are also allowed by SIM and can be specified by a DMSII mapping-options declaration.

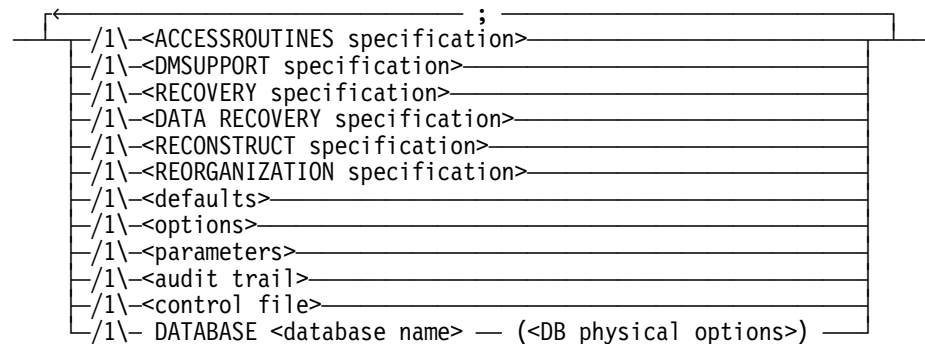
The following types of global mapping options can be declared:

- Software titles (such as ACCESSROUTINES and DMSUPPORT)
- Defaults (global, data set, and set defaults)
- Options (such as REAPPLYCOMPLETED and STATISTICS)
- Parameters (such as ALLOWEDCORE and SYNCWAIT)
- Audit trail attributes
- Control file attributes

### Syntax

The following diagram shows the syntax to use for declaring global mapping options:

**<global options>**



### Explanation

Except for the <database name> construct, the syntax and semantics of the options shown in the diagram are the same as in DMSII. All global mapping options specified with the syntax are essentially passed to DASDL “as is.” Therefore, refer to the *DMSII Data and Structure Definition Language (DASDL) Programming Reference Manual* for the detailed syntax and semantics of these options. The options shown in the diagram are explained briefly in the following table.

| Syntax Element                                      | Explanation  |
|---|--|
| <ACCESSROUTINES specification>                      | Specifies the title of the nontailored ACCESSROUTINES code file that the database is to use for general database operations.   |
| <DMSUPPORT specification>                           | Specifies the title of the tailored DMSUPPORT code file that is to be generated and used for general database operations.  |
| <RECOVERY specification>                            | Specifies the title of the nontailored RECOVERY code file that is to be used for database recovery operations.   |
| <DATA RECOVERY specification>                       | Specifies the title of the nontailored DATARECOVERY code file that is to be used for row recovery operations.  |
| <RECONSTRUCT specification>                         | Specifies the title of the tailored RECONSTRUCT code file that is to be used to initiate row recovery operations.  |
| <REORGANIZATION specification>                      | Specifies the title of the tailored REORGANIZATION code file that is to be generated and used for the initiation and support of the database reorganization process. |
| <defaults>  | Specifies global, data set, and set defaults, such as FAMILY and CHECKSUM options.   |
| <options>   | Specifies general database options, such as REAPPLYCOMPLETED and STATISTICS.   |
| <parameters>  | Specifies general database parameters, such as ALLOWEDCORE and SYNCWAIT.   |
| <audit trail>                                       | Specifies audit trail attributes, such as PACK and DUPLICATED options.   |
| <control file>                                      | Specifies control file attributes, such as PACK and SECURITYGUARD.   |
| DATABASE <database name><br>(<DB physical options>) | Specifies global database physical characteristics, such as a GUARDFILE specification, for the named database.   |

## Restrictions and Ramifications

Some restrictions on and ramifications of using global mapping options are described in the following paragraphs. The topics covered in these paragraphs relate to code file titles, defaults, options, parameters, audit trail attributes, control files, and database physical options.

### Code File Titles

It is recommended that the full titles of the ACCESSROUTINES, DMSUPPORT, RECOVERY, DATARECOVERY, RECONSTRUCT, and REORGANIZATION code files be specified, including the usercode and pack name for each file. This practice ensures that each file can be found by the appropriate software, regardless of the USERCODE or FAMILY specification of the process that is invoking the file. If a title is not specified for any of these code files, the DMSII default name is assumed for that file.

The SIM Utility automatically generates the tailored DMSUPPORT library with the correct title whenever it is needed. However, you must manually generate other tailored database code files, such as a RECONSTRUCT program or a DMINTERPRETER library. Refer to the *DMSII DASDL Reference Manual*, the *DMSII Utilities Operations Guide*, or the *InfoExec ADDS Operations Guide* for details on generating tailored database code files.

### Defaults

The following DMSII global default values are automatically assumed by SIM:

- CHECKSUM = TRUE
- PACK = <default pack: the primary family of the SIM Utility run>
- REBLOCK = TRUE
- DATA SET (LOCK TO MODIFY DETAILS = TRUE)

However, you can override these defaults by inserting an explicit <defaults> clause in a DMSII mapping-options declaration. You can also specify most of the other default options as well. Only the item defaults, which are ALPHA, REAL, and BOOLEAN, are not meaningful for SIM databases and are therefore not allowed.

### Options

The following DMSII global options are automatically assumed by SIM and cannot be overridden with a DMSII mapping-options declaration because they are required for every SIM database:

- ADDRESSCHECK
- AUDIT
- INDEPENDENTTRANS
- KEYCOMPARE

Note that when a SIM database schema is defined through the ADDS menu interface, the REAPPLYCOMPLETED parameter is automatically selected by ADDS. You can turn off REAPPLYCOMPLETED through a menu selection.

Any of the options allowed by DMSII can be specified, except for the RDSSTORE option, which is not allowed for SIM databases.

### Parameters

Any of the parameters allowed by DMSII can be specified for SIM databases. SIM does not specify any explicit values for parameters, thereby allowing DMSII to choose default values.

### Audit Trail Attributes

For audit trail attributes, the value *CHECKSUM = TRUE* is automatically assumed by SIM but can be overridden with a DMSII mapping-options declaration. All of the audit trail attributes allowed by DMSII are available for SIM databases.

### Control Files

Any of the control file attributes allowed by DMSII can be used for SIM databases. The usercode of the control file is the usercode under which all run-time database files reside, including the data sets, sets, and audit trails. The pack name of the control file defines the location of both the control file and the special data set called the SIM directory data set.

When a database is initially created by means of a SIM Utility ADD command, the usercode and the pack name specified within a <control file> clause of a DMSII mapping-options declaration interact with the database title given in the SIM Utility parameter as follows:

- The usercode specification in the <control file> clause is redundant with the usercode specification in the database title. If a usercode is specified in the <control file> clause and the database title, it must be the same in both places. If a usercode is not specified in either place, a CONTROL FILE ATTRIBUTES usercode declaration is not specified in the DASDL file; not specifying this pack name causes all run-time database files to reside under the usercode with which the SIM Utility is initiated.
- The pack name specification in the <control file> clause is redundant with the pack name specification in the database title. If a pack name is specified in the <control file> clause and the database title, it must be the same in both places. If a pack name is not specified in either place, a CONTROL FILE ATTRIBUTES pack name declaration is not specified in the DASDL file; not specifying this pack name causes the control file and the SIM directory data set to reside on the global default pack.



When a database schema is changed by means of a SIM Utility CHANGE command, the usercode and pack name specified within a <control file> clause of a DMSII mapping-options declaration are interpreted differently, as follows:

- The database title specified in the SIM Utility parameter is used to locate the current database, namely, the database that is to be changed.
- If a usercode or pack name is specified in a <control file> clause within the schema, the specification or specifications are interpreted as the new settings for the database. If neither option is specified, the value currently used by the control file of the database is used as the setting.
- If the control file pack name changes, the SIM Utility moves the control file and the SIM directory data set to the new location.
- If the control file usercode changes, the SIM Utility notes the change and passes the correct syntax to DMSII by means of the DASDL file. However, the SIM Utility does not actually move any database files to the new usercode because of the operational and security ramifications of moving files from one usercode to another. Instead, it issues a warning that the usercode change must be implemented “manually” after the SIM Utility completes processing.

If a SECURITYGUARD option is specified within a <control file> clause of the DMSII mapping-options declaration, evaluate the contents of the corresponding guard file. Refer to “Guard Files with SIM Databases” in this section for information on the use of SECURITYGUARD specifications.

### Database Physical Options

If a <DB physical options> clause is specified in a DMSII mapping-options declaration, the database name used in the specification must be the same as the database name given in the SIM Utility parameter. The only database physical option that is meaningful for a SIM database is the GUARDFILE option. Refer to “Guard Files with SIM Databases” in this section for information on the use of a database guard file.

### Performance Considerations

The criteria for setting global options are the same for SIM databases as they are for DMSII databases. Global-option settings that affect memory usage, disk usage, and auditing characteristics can influence the overall performance of the database. The settings for a number of global options critically affect the performance of a database and therefore must be given special attention when optimizing database performance. These global options are described in Table 9-1.

Table 9-1. Global Options That Critically Affect Performance

| Value Types            | Global Option Name             | Uses and Guidelines  |
|------------------------|--------------------------------|--|
| Defaults               | <disk/pack> specification      | The location of data sets and sets, the type of media used, and the number of physical paths to each pack used affects the performance of data access to those structures.   |
| Options                | REBLOCK specification          | This global option can be used to improve performance when specific database access patterns are known.  |
|                        | REAPPLYCOMPLETED               | The REAPPLYCOMPLETED option must be used when all completed transactions must be guaranteed to persist in the database, even after a halt/load recovery (such as when COMS is not being used to provide synchronized recovery). However, this option should be avoided when it is not required because it incurs a moderate amount of extra auditing overhead.                                       |
| Parameters             | ALLOWEDCORE                    | The ALLOWEDCORE parameter is critical; it seriously affects database performance. In general, set the ALLOWEDCORE parameter high enough so that buffer overlaying is minimized. The allowable value for ALLOWEDCORE is constrained by the amount of memory available on the system. Consider the tradeoff between available system memory and acceptable buffer overlaying when you set this option. |
|                        | CONTROLPOINT and SYNCPOINT     | These parameters affect the amount of auditing and, therefore, the amount of audit-related overhead that can be incurred by the database. You must weigh the value of these parameters against the recovery time of the database in the event of a halt/load or other database disruption.   |
|                        | OVERLAYGOAL                    | The OVERLAYGOAL parameter, in conjunction with the ALLOWEDCORE parameter, affects the amount of buffer overlaying that the database performs. The two parameters should be balanced so that efficient use of memory occurs without excessive buffer overlaying.  |
| Audit-trail attributes | AREAS, AREASIZE, and BLOCKSIZE | In some cases, these physical audit-trail attributes can be tuned for better performance during database auditing. For example, larger block sizes might reduce the amount of audit I/O time required by the database.   |

The performance of a particular database might be affected by global options other than those indicated in Table 9–1. Moreover, you might want to set some global options for operational considerations, for example, DMSUPPORT name and location, AUDIT TRAIL dump characteristics, and SECURITYGUARD specifications for physical structure security.

## Guard Files with SIM Databases

In DMSII, guard files can be used for two purposes:

- To control physical access to the files of the database
- To control logical access to the database and its logical databases

Guard files are employed for the first purpose through SECURITYGUARD specifications on the appropriate files. Guard files are employed for the second purpose through GUARDFILE specifications on the database as a whole or its logical databases. In the following paragraphs, these uses for guard files are considered with respect to SIM databases.

In general, considerations for using the SECURITYGUARD construct to provide physical protection for various DMSII files are the same for SIM databases as they are for DMSII databases. A SECURITYGUARD specification can be defined as a global default, a data set default, or a set/subset default. In each case, the associated guard file is inherited by the appropriate database files unless a more specific declaration is made. A specific SECURITYGUARD specification can be declared for the control file, audit trail, or any individual data set or set of the database.

If a default data set SECURITYGUARD specification is used, or if a global SECURITYGUARD specification is used that is to be inherited by data sets, then the corresponding guard file should include at least the following programs:

- All tailored and nontailored DMSII programs and utilities
- The SIM DMSII support library (typically named SYSTEM/SIM/DMSIISUPPORT)
- The SIM Utility (typically named SYSTEM/SIM/UTILITY)

The programs listed above should be included in the guard file because the SIM directory data set inherits this SECURITYGUARD specification and these programs all require physical access to this data set. For the tailored and nontailored DMSII programs and utilities, the guard file should be allowed read-write (RW) access. For the SIM DMSII support library and the SIM Utility program, read-only (RO) access by the guard file is sufficient.

### Example 1

For example, assume that the following DMSII mapping-options declaration is made:

```
DMSII-OPTIONS
  (DEFAULTS (SECURITYGUARD = *GUARDFILE/MYDB ON PACK)
   );
```

In this case, the guard file \*GUARDFILE/MYDB ON PACK should include specifications similar to the following:

```
PROGRAM *SYSTEM/DMUTILITY ON PACK      = RW;
PROGRAM *SYSTEM/DMRECOVERY ON PACK      = RW;
PROGRAM *SYSTEM/DMDATARECOVERY ON PACK  = RW;
PROGRAM *SYSTEM/ACCESSROUTINES ON PACK  = RW;
PROGRAM *RECONSTRUCT/MYDB ON PACK       = RW;
PROGRAM *SYSTEM/SIM/DMSIISUPPORT ON PACK = RO;
PROGRAM *SYSTEM/SIM/UTILITY ON PACK     = RO;
PROGRAM (DBA) REORGANIZATION/MYDB ON PACK = RW;
```

If a SECURITYGUARD specification is defined for the control file of the database, the corresponding guard file should include at least those specifications listed in Example 1. Inclusion of those specifications ensures that both DMSII and SIM can access the control file as needed. The corresponding guard file should also give read-only access to all users and application programs that open the database. The read-only access ability is necessary because the normal DMSII, and therefore SIM, database-open routine involves reading the control file of the database.

SIM requires no special physical access to the audit trails, data sets, or sets of the database, with the exception of the SIM directory data set, as discussed earlier. Therefore, considerations for attaching a specific SECURITYGUARD specification to any of these files are the same as they are for DMSII databases.

SIM databases do not use DMSII logical databases. Therefore, logical database GUARDFILE specifications are not allowed in a DMSII mapping-options declaration.

A GUARDFILE specification can be defined for the whole database in a <DB physical options> clause. This specification causes the associated guard file to control who can open the database as well as how the database can be opened. Native SIM databases (namely, those generated with SIM ODL as opposed to those generated by means of DMS.View or LINC.View) do not allow DMSII programs to open the database for update. Therefore, a GUARDFILE specification is not needed to protect a SIM database from damage by a DMSII program. However, a GUARDFILE specification might be desired to further control which DMSII programs can open the SIM database for inquiry.

SIM provides security control by allowing access declarations and permission declarations to be specified within the database schema. Therefore, use of a database guard file to exercise control over which SIM applications can open a SIM database is generally not necessary but can be done. Note that the program that appears to be opening a database is the running stack; this program is not a library that might be between the running stack and the DMSII ACCESSROUTINES code file. Therefore, only the titles of programs that open a SIM database on their own running stack need to be listed in the database guard file.

If a database guard file is specified, it should include at least the following:

- All tailored and nontailored DMSII programs and utilities
- The SIM DMSII support library
- The SIM Utility

In each case, the program should be allowed read-write (open-update) access.

### Example 2

For example, assume that the schema contains the following DMSII mapping-options declaration:

```
DMSII-OPTIONS
  (DATABASE MYDB (GUARDFILE = *GUARDFILE/MYDB/DBOPEN ON PACK)
  );
```

The guard file \*GUARDFILE/MYDB/DBOPEN ON PACK should contain specifications similar to the following:

```
PROGRAM *SYSTEM/DMUTILITY ON PACK      = RW;
PROGRAM *SYSTEM/DMRECOVERY ON PACK      = RW;
PROGRAM *SYSTEM/DMDATARECOVERY ON PACK  = RW;
PROGRAM *RECONSTRUCT/MYDB ON PACK       = RW;
PROGRAM *SYSTEM/SIM/DMSIISUPPORT ON PACK = RW;
PROGRAM *SYSTEM/SIM/UTILITY ON PACK     = RW;
PROGRAM (DBA) REORGANIZATION/MYDB ON PACK = RW;
```

Refer to the *DMSII DASDL Reference Manual* for more information on the use of the SECURITYGUARD and GUARDFILE constructs. Refer to the *Security Features Operations and Programming Guide* and the *Security Administration Guide* for information on the generation of guard files and for general information on security capabilities and techniques.

### Example Declaration of Global Mapping Options

In the following example, global mapping options are specified in a DMSII mapping-options declaration:

```
DMSII-OPTIONS
  (DMSUPPORT          = (PROD)DMSUPPORT/PAYROLLDB ON PACK;
   ACCESSROUTINES    = *SYSTEM/ACCESSROUTINES ON DMPACK;
   RECOVERY           = *SYSTEM/DMRECOVERY ON DMPACK;
   DEFAULTS
     (PACK             = PACK;
      SECURITYGUARD    = *GUARDFILE/PAYROLLDB ON PACK;
      DATA SET (PACK = DSPACK);
      SETS             (PACK = SETPACK);
     );
   PARAMETERS (ALLOWEDCORE = 500000);
   AUDIT TRAIL ATTRIBUTES (AREAS = 50);
   CONTROL FILE ATTRIBUTES (PACKNAME = CFPACK);
  );
```

In the example, the exact name and location are given for the DMSUPPORT, ACCESSROUTINES, and RECOVERY code files. A default pack name is given as PACK, but this is overridden by a more specific default of DSPACK for data sets, SETPACK for sets, and CFPACK for the control file. Only the audit trail inherits the pack name of PACK. A default SECURITYGUARD specification is given, which causes the corresponding guard file to be attached to all structure files as well as to the control and audit trail files. The ALLOWEDCORE parameter of the database is set at 500,000 words, and audit trail files are restricted to a maximum of 50 areas each.



### TIMESTAMP Scheme

The TIMESTAMP scheme is the default surrogate scheme for all base classes. Since TIMESTAMP is the default, it need not be specified in the database schema file. However, you might wish to specify it, for instance, to document its use. The TIMESTAMP scheme causes SIM to generate a surrogate attribute for the corresponding base class and to map it to a unique item in the disjoint data set for that class. This unique item is termed the surrogate item, and it is spanned by an INDEX SEQUENTIAL, NO DUPLICATES set called the surrogate set. The value stored in a TIMESTAMP surrogate item is essentially a number that is continuously incremented for each new entity. The surrogate value of a deleted entity is never reused.

If the hierarchy of the base class has additional disjoint data sets, each one also contains a surrogate item and a corresponding surrogate set. As an entity is extended into various subclasses in the hierarchy, the surrogate value of the entity is copied to the appropriate disjoint data sets (much like a primary key is copied to foreign key locations as a means of creating relationships). This process allows any record for the entity to be found by means of the surrogate value of the entity.

The TIMESTAMP scheme has the following advantages:

- The surrogate attribute is automatically generated and maintained, and it is completely independent from user-defined attributes. This independence ensures that the identity of each entity remains unique as long as that entity exists in the database, regardless of changes that might occur to the user-defined attributes for the entity.
- SIM performs some degree of table optimization when selecting the INDEX SEQUENTIAL sets, resulting in fairly good performance when creating and retrieving entities.

The primary disadvantage of the TIMESTAMP scheme is that it adds a set to each disjoint data set in the hierarchy. This set incurs a slight overhead when entities are created and deleted, although it has only a negligible impact when entities are modified. When a data set record is needed for an entity with a given surrogate value (for example, when an EVA to the entity is traversed), a binary search is performed on the INDEX SEQUENTIAL surrogate set. This binary search results in at least one I/O to the data set. The number of I/Os on the set depends on which of its tables are in memory.

The performance characteristics associated with the TIMESTAMP scheme are generally very reasonable for most applications. Furthermore, this option has none of the disadvantages of the attribute surrogate scheme. Therefore, the TIMESTAMP scheme is generally recommended for most databases. When performance is critical, however, better update and retrieval performance can be obtained by using the attribute scheme for mapping surrogates. The attribute scheme is described under the heading “Attribute Scheme” later in this section.

Use of the TIMESTAMP scheme is shown in the following example.



**Example**

Consider the following schema:

```

CLASS Employee
  (emp-no:  NUMBER [5], REQUIRED, UNIQUE;
  );

DMSII-OPTIONS
  (CLASS Employee (SURROGATE = TIMESTAMP)
  );

```

The Employee class is basically mapped to the following DMSII structures:

```

EMPLOYEE DATASET
  (SIM-SURR-1  FIELD (48) REQUIRED;
   EMP-NO      NUMBER (5);
  );

SIM-SET-1 SET OF EMPLOYEE
  KEY IS (SIM-SURR-1), NO DUPLICATES;

SIM-SET-2 SET OF EMPLOYEE
  KEY IS (EMP-NO), NO DUPLICATES KEYCHANGEOK;

```

Note that SIM-SURR-1 is the surrogate item and SIM-SET-1 is the surrogate set. Also note that since the DVA emp-no is specified as UNIQUE, another set, SIM-SET-2, is generated to enforce the uniqueness constraint.

**Attribute Scheme**

The attribute scheme is an alternate surrogate scheme that can be specified for a base class. This scheme can be used only by a class that contains a user-declared, single-valued data-valued attribute (DVA) that meets all of the following conditions:

- It is specified as REQUIRED and UNIQUE.
- Its type is neither SUBROLE nor COMPOUND.
- If its type is NUMBER, it has 12 digits of precision or less.
- If its type is STRING, its length is relatively small (for example, 12 bytes or less). There are no explicit length limits, but large strings degrade performance.
- Its value never needs to be changed once an entity is created.

When a DVA meets all of these conditions, you can designate it as the surrogate attribute of the class by using its name as the <attr name> parameter within the SURROGATE option. When this option is used, the surrogate attribute that is normally generated under the TIMESTAMP scheme and the corresponding surrogate item and surrogate set are eliminated. Instead, the item and set that are generated on behalf of the DVA are used as the surrogate item and surrogate set for the class. This arrangement has two distinct advantages:

- The elimination of the extra set results in better performance during insertions and deletions of entities.
- Depending on the manner in which the surrogate set is implemented, better retrieval performance can be achieved.

In the following situation, the surrogate set is implemented in a way that can achieve better performance. If the DVA is the only key attribute of a user-declared index, the index is mapped to the surrogate set. Consequently, mapping options can be declared on the index. These mapping options are then applied to the set, thereby providing user control over the surrogate set. (In the TIMESTAMP scheme, the surrogate set cannot be controlled through user-specified options.) Refer to “Data Set Options for a Base Class” in this section for an example that illustrates this situation.

### Example

Assume that the example schema presented under “TIMESTAMP Scheme” was declared using the attribute scheme rather than the TIMESTAMP scheme, as follows:

```
CLASS Employee
  (emp-no:  NUMBER [5], REQUIRED, UNIQUE;
  );

DMSII-OPTIONS
  (CLASS Employee (SURROGATE = emp-no)
  );
```

In this case, the class would be mapped to the following DMSII structures:

```
EMPLOYEE DATASET
  (EMP-NO  NUMBER (5) REQUIRED;
  );

SIM-SET-1 SET OF EMPLOYEE
  KEY IS (EMP-NO), NO DUPLICATES;
```

As shown, only one set is needed for the EMPLOYEE data set. This set is used to enforce the uniqueness of the emp-no attribute as well as to provide entity access by a surrogate value. The attribute scheme affords a significant enough performance improvement for its use to be recommended in cases where performance is of utmost importance.

Some drawbacks in using the attribute scheme are the following:

- In a pure interpretation of the semantic data model, surrogates should be maintained by the system and be independent of all user-visible attributes. The attribute scheme makes the surrogate attribute both user-visible and user-controlled.
- SIM normally allows all attributes to be modified, even if their types are REQUIRED and UNIQUE. However, a DVA that is used as a surrogate attribute cannot change values once an entity has been inserted.
- Since SIM is not generating surrogate values, these values are completely under the control of the user. With the TIMESTAMP scheme, SIM never reuses old surrogate values. With the attribute scheme, however, it cannot be guaranteed that old surrogate values are never reused. Whether this aspect of the attribute scheme is a drawback depends on the application semantics for the reuse of old surrogate values.

Refer to “Data Set Options for a Base Class” in this section for an example of how the attribute scheme can be used in conjunction with a data set option to achieve better performance than either type of option would provide separately.

## Data Set Options for a Base Class

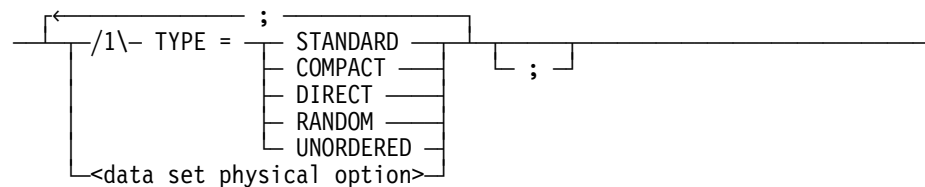
A base class is mapped by default to a variable-format data set of the type STANDARD. SIM specifies no physical options about the data set other than the data set type. DMSII chooses default values for the remaining data set options.

By specifying data set mapping options for a base class in the schema file, you can change the type of data set used as well as the data set physical options.

### Syntax

The following diagram shows the syntax to use for specifying data set mapping options:

**<data set options>**



### Explanation

The syntax elements shown in the diagram are explained in the following table:

| Syntax Element             | Explanation   |
|----------------------------|---|
| TYPE = STANDARD            | Specifies that the data set is to be of the type STANDARD. This is the default data set type.   |
| TYPE = COMPACT             | Specifies that the data set is to be of the type COMPACT.   |
| TYPE = DIRECT              | Specifies that the data set is to be of the type DIRECT.  |
| TYPE = RANDOM              | Specifies that the data set is to be of the type RANDOM.  |
| TYPE = UNORDERED           | Specifies that the data set is to be of the type UNORDERED.<br><br>Refer to the <i>DMSII DASDL Reference Manual</i> for descriptions of STANDARD, COMPACT, DIRECT, RANDOM, and UNORDERED data sets. |
| <data set physical option> | Specifies physical options for the data set, such as POPULATION or PACK. Refer to the <i>DMSII DASDL Reference Manual</i> for details on the syntax and semantics of <data set physical option>.    |

### Examples

The following is an example specification of data set mapping options in a schema file:

```
CLASS Employee
  (emp-no:  NUMBER [5], REQUIRED, UNIQUE;
   );

DMSII-OPTIONS
  (CLASS Employee
    (DATASET-OPTIONS
      (POPULATION = 100000;
       BLOCKSIZE  = 208 RECORDS;
      )
    )
  );
```

This schema causes the designated POPULATION and BLOCKSIZE specifications to be attached to the data set, thereby providing increased capacity and improved blocking for the data set.

The criteria for optimizing a DMSII data set are the same for SIM as they are for DMSII. Therefore, normal rules for reducing wasted space, improving I/O performance, and meeting other criteria can be used in specifying data set mapping options in a schema file.

The use of data set options to set the POPULATION parameter for the class is highly recommended, and sometimes mandatory, since the DMSII default limit of 10,000 records is often an insufficient capacity for a data set.

In some cases, the SURROGATE option can be used in conjunction with data set options to achieve better performance than that provided by either type of option separately. For example, consider the following schema:

```
CLASS Employee
  (emp-no:  NUMBER [5], REQUIRED, UNIQUE;
  );
INDEX Employee-numbers ON Employee
  (emp-no) UNIQUE;

DMSII-OPTIONS
  (CLASS Employee
    (SURROGATE = emp-no;
     DATASET-OPTIONS (TYPE = DIRECT);
    );
  INDEX Employee-numbers
    (SET-OPTIONS (TYPE = ACCESS))
  );
```

This schema uses emp-no as the surrogate attribute for the Employee base class. The schema maps Employee to a data set of the type DIRECT, and uses the user-defined index Employee-numbers mapped to an access as the surrogate index for the Employee base class. (Refer to “Index Mapping Options” later in this section).

The resulting DASDL syntax would be the following:

```
EMPLOYEE DIRECT DATASET
  (EMP-NO NUMBER (5) REQUIRED
  );

EMPLOYEE-NUMBERS ACCESS TO EMPLOYEE
  KEY IS (EMP-NO), NO DUPLICATES;
```

This mapping arrangement is highly efficient because a single physical structure is used for the class, and access through a surrogate value results in direct record accesses.

# Attribute Mapping Options

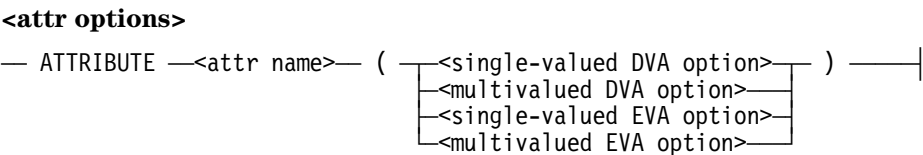
Each SIM attribute is mapped into DMSII constructs that fulfill the storage and integrity requirements of the attribute. Some attributes have a variety of mapping techniques that reflect tradeoffs in performance and space usage. The manner in which an attribute can be mapped depends on whether the attribute is a class attribute, a DVA, or an EVA and on the following factors:

- If the attribute is a class attribute or a DVA: the type of the attribute (for example, INTEGER or STRING) and the allowable values (for example, length and subrange specifications)
- If the attribute is a DVA or an EVA: the cardinality of the attribute (for example, single valued or multivalued) and other attribute options (for example, UNIQUE, DISTINCT, and MAX)
- If the attribute is an EVA: the surrogate scheme used by the hierarchy over which the EVA ranges (for example, the TIMESTAMP scheme or the attribute scheme)

For each attribute, SIM chooses a default mapping that represents a reasonable balance between performance and space usage. For many applications, better performance and/or space usage can be obtained for an attribute by using attribute mapping options. Also, attribute mapping options are sometimes needed to override DMSII defaults (such as the POPULATION specification for a data set) with more suitable values.

## Syntax

The following diagram shows the syntax to use for specifying attribute mapping options:



## Explanation

The syntax elements shown in the diagram are explained briefly in the following table:

| Syntax Element             | Explanation   |
|----------------------------|---|
| ATTRIBUTE <attr name>      | Identifies the DVA or the EVA to which the attribute options apply. |
| <single-valued DVA option> | Specifies an option that applies to a single-valued DVA.            |
| <multivalued DVA option>   | Specifies an option that applies to a multivalued DVA.              |
| <single-valued EVA option> | Specifies an option that applies to a single-valued EVA.            |
| <multivalued EVA option>   | Specifies an option that applies to a multivalued EVA.              |

The default mapping and mapping options for each type of attribute are presented under the following headings:

- Class Attribute Mapping
- Single-Valued DVA Mapping
- Multivalued DVA Mapping
- EVA Mapping

## Class Attribute Mapping

The default mapping of class attributes is described in the following paragraphs. Currently, there are no user-specifiable mapping options for class attributes.

All class attributes are mapped to a special data set called the class-attribute data set. This data set is a variable-format data set of the type STANDARD. Therefore, this data set allows each variable-format record type to be used only once. All class attributes for a given class or a given subclass are mapped to a single variable-format record.

Each class attribute is mapped to two items within the appropriate variable-format record:

- A null bit item, which indicates when the attribute is null
- A data item, which contains the attribute value when it is not null

Additional items might be generated for a class attribute in special cases, but the null bit item and data item are always generated.

### Example

Consider the following schema:

```
CLASS Person
  CLASS-ATTRIBUTES
    (next-person-id      : INTEGER;
    )
  (...
  );

SUBCLASS Employee OF Person
  CLASS-ATTRIBUTES
    (next-employee-id   : INTEGER;
    last-hire-date      : DATE;
    maximum-salary      : NUMBER [8,2];
    )
  (...
  );
```

The class attributes in this schema would be mapped to the following DMSII class-attribute data set:

```
SIM-CLASSATTS-1 DATA SET
  (SIM-RECORD-TYPE      RECORD TYPE (254);
   SIM-DUMMY-ITEM      ALPHA (1) REQUIRED INITIALVALUE " ";
  )
1:                      % class attributes for Person
  (SIM-CTRLFLD          FIELD
   (NULL-NEXT-PERSON-I  BOOLEAN;
  );
  NEXT-PERSON-ID        REAL ($11);
  )
2:                      % class attributes for Employee
  (SIM-CTRLFLD-2        FIELD
   (NULL-NEXT-EMPLOYEE  BOOLEAN;
    NULL-LAST-HIRE-DAT  BOOLEAN;
    NULL-MAXIMUM-SALAR  BOOLEAN;
  );
  NEXT-EMPLOYEE-ID      REAL ($11);
  LAST-HIRE-DATE        FIELD (28);
  MAXIMUM-SALARY        NUMBER (8,2);
  );

SIM-CLATT-SET-1 SET OF SIM-CLASSATTS-1
  KEY IS (SIM-RECORD-TYPE), NO DUPLICATES;
```

As shown, the class attributes of Person are mapped to variable format 1, and the class attributes of Employee are mapped to variable format 2. The set allows the class attributes of a given class or a given subclass to be retrieved quickly. Moreover, it ensures that each variable-format record occurs only once. This mapping scheme offers good performance and space usage, and it minimizes the potential for bottlenecks that can occur if class attributes are accessed frequently.

## Single-Valued DVA Mapping

The mapping of single-valued DVAs is described under the following headings:

- Default Single-Valued DVA Mapping
- Single-Valued DVA Mapping Options

### Default Single-Valued DVA Mapping

Each single-valued DVA is mapped to two items in the data set of its owner:

- A null bit item, which indicates when the DVA is null
- A data item, which contains the value for the DVA when it is not null

Additional items might be generated for a DVA in special cases, but the null bit item and data item are always generated.



**Example**

Consider the following schema:

```
CLASS Person
  (name: STRING [20]
  );
```

The single-valued DVA in this schema would be mapped as follows:

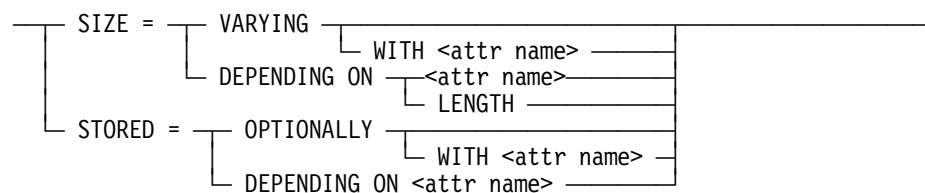
```
PERSON DATA SET
  (SIM-CTRLFLD      FIELD
   (NULL-NAME      BOOLEAN;
   );
   NAME            ALPHA (20);
  );
```

**Single-Valued DVA Mapping Options**

The mapping options that can be used for a single-valued DVA are presented in the following paragraphs.

**Syntax**

The following diagram shows the syntax to use for specifying a single-valued DVA mapping option:

**<single-valued DVA option>****Explanation**

The SIZE and STORED options enable you to add a storage compaction option to the data item for the DVA. These options translate into corresponding SIZE and STORED options provided by DMSII. Consequently, only one of these options can be chosen for a single-valued DVA, and it can be chosen only when the class or subclass that owns the DVA has been mapped to a COMPACT data set. Adding a compaction option can result in the data item for the DVA using less storage space or, under certain circumstances, not being stored at all.

Each possible compaction option is first shown and then described in the following paragraphs. After all of these options are described, an example is presented that uses several of the compaction options.

### SIZE = VARYING

This compaction option is allowed only for DVAs of the type STRING (string DVAs). It causes the data item for the DVA, which is an ALPHA item, to be declared in DMSII with the *SIZE VARYING* option. This declaration causes the data item to be stored without trailing blanks, thereby potentially reducing the amount of space that it requires within each data set record.

### SIZE = VARYING WITH <attr name>

This compaction option is similar to the *SIZE = VARYING* option, except that the data item for the string DVA is declared in DMSII with the *SIZE VARYING WITH <item name>* option. The <item name> parameter given to DMSII is the data item of the specified <attr name> parameter, which is called the size attribute. The size attribute must meet the following conditions:

- It must be an attribute of the same class or subclass as the string DVA.
- It must be a single-valued DVA declared as READONLY.
- Its type must be NUMBER, and its precision must be large enough to hold the maximum possible length of the string.
- It must not have a subrange condition.
- It cannot be specified as UNIQUE, and it cannot be a key of an index.

After an entity possessing a string DVA that uses this compaction option is stored (inserted or modified), you can inquire on the size attribute to determine the number of characters actually stored within the data set record for the string DVA. When the string DVA is null, the size attribute is also null.

### SIZE = DEPENDING ON <attr name>

This compaction option is similar to the *SIZE = VARYING* and *SIZE = VARYING WITH <attr name>* options, except that it causes the data item for the string DVA to be declared in DMSII with the *SIZE DEPENDING ON <item name>* option. The <item name> parameter given to DMSII is the data item of the <attr name> parameter that you specify, which is the size attribute. The size attribute must be a single-valued DVA belonging to the same class or subclass as the string DVA, and its type must be an unsigned NUMBER.

Using this compaction option enables you to determine how much of the string DVA is actually stored for each entity. When an entity is inserted or modified, the value of the size attribute is tested, and only the indicated number of leading characters of the string DVA is actually stored within the corresponding data set record.

**SIZE = DEPENDING ON LENGTH**

This compaction option is similar to the *SIZE = DEPENDING ON <attr name>* option, except that it is allowed only for string DVAs declared as VARIABLE. SIM automatically generates a size attribute for string DVAs that are declared as VARIABLE.

Using this compaction option causes the size attribute to be used for determining how much of the DVA is stored. Specifically, the data item for the DVA is stored truncated to the assigned length of the DVA. For example, if the DVA were given the value ABC for a particular entity, only three characters would be stored in the corresponding data set record for the DVA.

**STORED = OPTIONALLY**

This compaction option is allowed for all DVA types except SUBROLE, compound, and BOOLEAN. For string DVAs, the option is allowed only when one of the SIZE compaction options has not been used. This option causes the data item for the DVA to be declared in DMSII with the *STORED OPTIONALLY* option. This declaration causes the data item to be stored only when its value does not equal the DMSII null value, which is always all-bits-on for SIM databases. The value of the data item equals all-bits-on in the following two cases:

- When the attribute is null in the SIM sense, which means that the null bit of the attribute is set to TRUE
- When all-bits-on is a valid data value for the DVA, and the DVA is assigned that value (for example, when a string is assigned a value of all-bits-on)

In summary, this compaction option allows a savings in space when a DVA is null and when its value equals all-bits-on.

**STORED = OPTIONALLY WITH <attr name>**

This compaction option is similar to the *STORED = OPTIONALLY* option, except that it causes the data item for the single-valued DVA to be declared in DMSII with the *STORED OPTIONALLY WITH <item name>* option. The item name given to DMSII is the data item of the storage attribute specified as the <attr name> parameter.

The storage attribute must meet the following conditions:

- It must be a single-valued DVA belonging to the same class or subclass as the single-valued DVA on which the compaction option is declared.
- Its type must be BOOLEAN.
- It must be declared as READONLY.
- It must not be specified as UNIQUE, and it must not be a key of an index.

When an entity possessing a single-valued DVA that uses this compaction option is stored (inserted or modified), an inquiry can be made on the storage attribute to determine if the single-valued DVA was actually stored within the corresponding data set record.

**STORED = DEPENDING ON <attr name>**

This compaction option is similar to the *STORED = OPTIONALLY* and *STORED = OPTIONALLY WITH <attr name>* options, except that it causes the data item for the single-valued DVA to be declared in DMSII with the *STORED DEPENDING ON <item name>* option. The item name given to DMSII is the data item of the storage attribute specified by the <attr name> parameter. The storage attribute must be a single-valued DVA belonging to the same class or subclass as the single-valued DVA on which the compaction option is declared, and its type must be BOOLEAN. This compaction option causes the single-valued DVA to be stored only when the storage attribute is TRUE.

### Example of Compaction Options

The following example uses several of the compaction options described in the preceding paragraphs.

Consider the following schema:

```
CLASS Person
  (name:          STRING [20];
   job-title:     STRING [25];
   job-title-length:  NUMBER [2], READONLY;
   job-description:  STRING [1000] VARIABLE;
   dependent-info:  STRING [100];
   valid-dep-info:  BOOLEAN;
  );

DMSII-OPTIONS
  (CLASS Person
   (DATASET-OPTIONS (TYPE = COMPACT);
    ATTRIBUTE name          (SIZE = VARYING);
    ATTRIBUTE job-title      (SIZE = VARYING WITH
                             job-title-length);
    ATTRIBUTE job-description (SIZE = DEPENDING ON LENGTH);
    ATTRIBUTE dependent-info (STORED = OPTIONALLY WITH
                             valid-dep-info);
   )
  );
```

The Person class defined in this schema would be mapped into the following DMSII DASDL syntax:

```

PERSON COMPACT DATA SET
  (SIM-CTRLFLD      FIELD
   (NULL-NAME       BOOLEAN;
    NULL-JOB-TITLE  BOOLEAN;
    NULL-JOB-DESCRIPT1  BOOLEAN;
    NULL-DEPENDENT-INF  BOOLEAN;
    NULL-VALID-DEP-INF  BOOLEAN;
    VALID-DEP-INFO    BOOLEAN;
   );
  NAME              ALPHA (20) SIZE VARYING;
  JOB-TITLE-LENGTH  NUMBER (2);
  JOB-TITLE          ALPHA (25) SIZE VARYING WITH
                     JOB-TITLE-LENGTH;
  SZCTRL-JOB-DESCRIP  NUMBER (4);
  JOB-DESCRIPTION     ALPHA (1000) SIZE DEPENDING ON
                     SZCTRL-JOB-DESCRIP;
  DEPENDENT-INFO      ALPHA (100) STORED DEPENDING ON
                     VALID-DEP-INFO;
);

```

In this example, the Person class is mapped to a compact data set and uses single-valued DVA options as follows:

- Name is a string DVA that uses the *SIZE = VARYING* option, which causes trailing blanks to be truncated when the DVA is stored in a data set record.
- Job-title is a string DVA that uses the *SIZE = VARYING WITH <attr name>* option, which causes the stored, truncated length of the DVA to be reflected in the job-title-length attribute.
- Job-description is a VARIABLE string DVA that uses the *SIZE = DEPENDING ON LENGTH* option, which causes only the assigned characters of the DVA to be stored in the data set record.
- Dependent-info is a string DVA that uses the *STORED = OPTIONALLY WITH <attr name>* option. This option causes dependent-info to be stored only when it is nonnull and non-all-bits-on. Moreover, the option causes the attribute valid-dep-info to reflect whether a value has been stored for the attribute dependent-info.

### Multivalued DVA Mapping

The mapping of multivalued DVAs is described under the following headings:

- Default Multivalued DVA Mapping
- Multivalued DVA Mapping Options

#### Default Multivalued DVA Mapping

There are three basic techniques or mapping types with which a multivalued DVA can be mapped:

- DISJOINT-DATASET
- OCCURRING-ITEM
- EMBEDDED-DATASET

By default, most multivalued DVAs are assigned the DISJOINT-DATASET mapping type, which causes a multivalued nonsubrole DVA to be mapped to a disjoint data set with one or two spanning sets.

The only other default mapping type for multivalued DVAs is OCCURRING-ITEM, which causes a multivalued DVA to be mapped to an occurring item within the data set of its owner. The OCCURRING-ITEM mapping type is automatically used when a multivalued DVA meets certain conditions.

The third mapping type, EMBEDDED-DATASET, is not used by default, but it is available as an option. This mapping type causes a multivalued DVA to be mapped to an embedded data set within the disjoint data set of its owner. Depending on the attribute options for the multivalued DVA, the embedded data set might or might not be spanned by an embedded set.

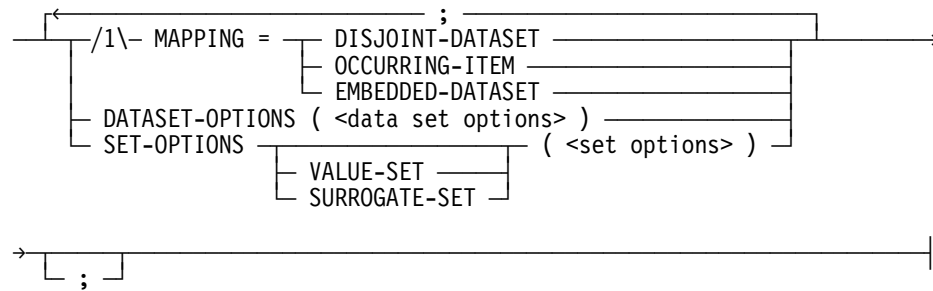
Each possible multivalued DVA mapping type is described under the next heading, “Multivalued DVA Mapping Options.” Note that multivalued subrole DVAs require special mapping techniques other than the three techniques described in the preceding paragraphs. Moreover, the mapping of multivalued subrole DVAs cannot be influenced by user options. Therefore, mapping techniques for multivalued subrole DVAs are not discussed in this section.

#### Multivalued DVA Mapping Options

Multivalued DVA mapping options enable you to control the mapping type of the multivalued DVA and, when applicable, physical characteristics of its mapped data set and sets. Multivalued DVA mapping options are allowed on multivalued DVAs whose type is not SUBROLE.

**Syntax**

The following diagram shows the syntax to use for specifying a multivalued DVA mapping option:

**<multivalued DVA option>****Explanation**

The syntax elements shown in the diagram are explained briefly in the following table and are explained in detail in the paragraphs that succeed the table.

| Syntax Element                            | Explanation  |
|---|--|
| MAPPING = DISJOINT-DATASET                | Specifies that the multivalued DVA is to be mapped to a disjoint data set.                               |
| MAPPING = OCCURRING-ITEM                  | Specifies that the multivalued DVA is to be mapped to an occurring item.                                 |
| MAPPING = EMBEDDED-DATASET                | Specifies that the multivalued DVA is to be mapped to an embedded data set.                              |
| DATASET-OPTIONS (<data set options>)      | Specifies physical options for the disjoint or embedded data set to which the multivalued DVA is mapped. |
| SET-OPTIONS (<set options>)               | Specifies physical options for the value set of the multivalued DVA.                                     |
| SET-OPTIONS VALUE-SET (<set options>)     | Specifies physical options for the value set of the multivalued DVA.                                     |
| SET-OPTIONS SURROGATE-SET (<set options>) | Specifies physical options for the surrogate set of the multivalued DVA.                                 |

### MAPPING Option

The MAPPING option controls the basic mapping type of the multivalued DVA. As previously indicated under “Default Multivalued DVA Mapping,” SIM automatically chooses either the DISJOINT-DATASET mapping type or, under certain conditions, the OCCURRING-ITEM mapping type. Additionally, SIM offers the EMBEDDED-DATASET mapping type for multivalued DVAs. The MAPPING option enables you to choose a nondefault mapping type or to declare explicitly a default mapping type for documentation purposes. Each of these three mapping types is explained in the following paragraphs.

#### MAPPING = DISJOINT-DATASET

By default, most multivalued DVAs are mapped to a separate, disjoint, standard data set that is logically linked to the data set of the owner of the DVA with a foreign-key relationship. This is the DISJOINT-DATASET mapping type. The disjoint data set is spanned by at least one INDEX SEQUENTIAL set, called the surrogate set, which gives access to multivalued DVA values by means of the surrogate value of a specific entity. The surrogate set is also used to enforce the DISTINCT option when it is specified.

For example, consider the following schema:

```
CLASS Person
  (children:  STRING [15], MV (DISTINCT);
  );
```

The multivalued DVA named children in this schema would be mapped to the following disjoint data set:

```
CHILDREN DATA SET
  (SIM-SURR-1      FIELD (48);
   DATA-CHILDREN  ALPHA (15);
  );

SIM-INDEX-1 SET OF CHILDREN
  KEY IS (SIM-SURR-1, DATA-CHILDREN), NO DUPLICATES;
```

Each record in the CHILDREN data set holds one value for the children DVA. SIM-SURR-1 holds a copy of the Person surrogate value to which each record is related, and DATA-CHILDREN holds the CHILDREN value for each record. The surrogate set SIM-INDEX-1 provides access to children values by means of the surrogate value of each Person entity. Since DATA-CHILDREN has been added to SIM-INDEX-1 as a secondary key and the set is declared with NO DUPLICATES, the set also enforces the DISTINCT option. If children were not declared as DISTINCT, the secondary key would not be present and the set would allow duplicates.



In some cases, a second INDEX SEQUENTIAL set, called a value set, is added to span the data set of a multivalued DVA. For instance, a value set is added when the multivalued DVA is declared as UNIQUE. If the multivalued DVA named children in the preceding example were declared as UNIQUE instead of DISTINCT, it would be mapped to the following disjoint data set instead:

```
CHILDREN DATA SET
  (SIM-SURR-1      FIELD (48);
   DATA-CHILDREN  ALPHA (15);
  );

SIM-INDEX-1 SET OF CHILDREN
  KEY IS (SIM-SURR-1), DUPLICATES;

SIM-INDEX-2 SET OF CHILDREN
  KEY IS (DATA-CHILDREN), NO DUPLICATES;
```

SIM-INDEX-1 is the surrogate set, which provides access by a surrogate value. SIM-INDEX-2 is the value set that enforces the UNIQUE constraint. If an index were declared on Person with children as its key, this index would be used for the value set instead of SIM-INDEX-2. In other words, a user-declared index always becomes a value set, and a value set is always used to enforce the UNIQUE option when specified.

The main advantage of the DISJOINT-DATASET mapping type for a multivalued DVA is that it is very flexible. You can add or remove attribute options such as DISTINCT, UNIQUE, and MAX by means of a CHANGE schema. Access to multivalued DVA values is fairly efficient, although the extra structures required induce some additional processing overhead.

You might want to specify the DISJOINT-DATASET option to override a case in which a multivalued DVA is normally mapped to an occurring item. That is, in a situation where SIM would normally map a multivalued DVA using the OCCURRING-ITEM mapping, you can force the multivalued DVA to be mapped with the DISJOINT-DATASET option instead. This action avoids, for example, the change restrictions placed on a multivalued DVA mapped to an occurring item. These change restrictions are described under "MAPPING = OCCURRING-ITEM."

#### MAPPING = OCCURRING-ITEM

The other default mapping type that SIM uses for multivalued DVAs is OCCURRING-ITEM. This mapping type is automatically chosen for a multivalued DVA when it meets the following conditions:

- It is not specified as DISTINCT or UNIQUE.
- It is not the key of a user-declared index.
- It has a MAX limit of 1023 or less.

When these basic conditions are met, the multivalued DVA is mapped to an occurring item within the data set of its owner.

For example, consider the following schema:

```
CLASS Person
  (children:  STRING [15], MV (MAX 10);
  );
```

The multivalued DVA named children is mapped as follows:

```
PERSON DATA SET
  (CHILDREN
    (SIM-CTRLFLD      FIELD
      (NULL-CHILDREN  BOOLEAN;
    );
    DATA-CHILDREN    ALPHA (15);
  )  OCCURS 10 TIMES;
);
```

As shown, children is mapped to an occurring item within the PERSON data set such that each PERSON record can hold up to the maximum number of CHILDREN values possible.

The advantage of this clustered mapping technique is that all CHILDREN values reside on the same record as their owning PERSON entity. Consequently, no extra I/O is needed to retrieve or update CHILDREN values.

The disadvantage of the OCCURRING-ITEM mapping type is that the multivalued DVA is heavily restricted in terms of its attribute options and the manner in which they may be changed. Specifically, the following restrictions apply:

- The multivalued DVA cannot be specified as DISTINCT or UNIQUE, and these options cannot be added to it by means of a CHANGE schema.
- The MAX limit of the multivalued DVA can be increased by means of a CHANGE schema, but it cannot be increased beyond 4095 and it cannot be decreased.
- The multivalued DVA cannot be a key of a user-specified index, and it cannot be made a key by means of a CHANGE schema.

The higher performance of occurring items often outweighs these restrictions, thereby making this mapping type attractive when it can be used. Therefore, the OCCURRING-ITEM mapping type is a default when feasible.

There is one case in which you can specify the OCCURRING-ITEM option when it is not the default mapping type. This case is when a multivalued DVA meets all of the basic conditions above, but its MAX limit is between 1024 and 4095, inclusive. In this case, you can specify the OCCURRING-ITEM mapping type and obtain the better performance offered by this technique.

**MAPPING = EMBEDDED-DATASET**

SIM never automatically chooses the EMBEDDED-DATASET mapping type for a multivalued DVA, but you can specify it as an option. This mapping type can be specified for any multivalued DVA that is not specified as UNIQUE and that is not a key of a user-declared index.

The EMBEDDED-DATASET mapping type causes the multivalued DVA to be mapped to an embedded data set within the disjoint data set of its owner. If the multivalued DVA is not specified as DISTINCT, the corresponding data set is UNORDERED and is not spanned by a set. If the multivalued DVA is specified as DISTINCT, the corresponding data set is STANDARD and is spanned by an embedded, INDEX SEQUENTIAL value set that specifies NO DUPLICATES, thereby enforcing the DISTINCT option.

For example, consider the following schema:

```
CLASS Person
  (children:  STRING [15], MV (DISTINCT);
  );

DMSII-OPTIONS
  (CLASS Person
    (ATTRIBUTE children (MAPPING = EMBEDDED-DATASET))
  );
```

The multivalued DVA named children is mapped as follows:

```
PERSON DATA SET
  (CHILDREN STANDARD DATA SET
    (SIM-CTRLFLD      FIELD
      (SIM-VALID-BIT  BOOLEAN;
    );
    DATA-CHILDREN    ALPHA (15);
  );
  SIM-INDEX-1 SET OF CHILDREN
    KEY IS (DATA-CHILDREN), INDEX SEQUENTIAL, NO DUPLICATES;
);
```

As shown, the children DVA is mapped to an embedded standard data set within the PERSON data set. The embedded set SIM-INDEX-1 provides access to CHILDREN records, and it enforces the DISTINCT option by ensuring that no PERSON record has duplicate CHILDREN values.

The advantages of the EMBEDDED-DATASET mapping type are the following:

- It offers somewhat better performance than does the DISJOINT-DATASET mapping type.
- It does not possess all of the restrictions of the OCCURRING-ITEM mapping type.

The EMBEDDED-DATASET mapping type is especially efficient when the multivalued DVA is not DISTINCT because each master record points to embedded blocks of contiguous records. Therefore, few I/Os are needed to retrieve all embedded records for a particular master record.

However, there are several disadvantages of the EMBEDDED-DATASET mapping type. These disadvantages consist of the following restrictions:

- A multivalued DVA mapped to an embedded data set cannot be added to an existing data set. Therefore, such a multivalued DVA cannot be added to an existing base class or subclass. Moreover, it cannot be added to a new subclass that is to be mapped to a variable format of an existing data set.
- The multivalued DVA cannot be specified as UNIQUE, and the UNIQUE option cannot be added by means of a CHANGE schema.
- The multivalued DVA cannot be a key of a user-declared index, and it cannot be made a key by means of a CHANGE schema.
- If the disjoint data set in which the multivalued DVA is mapped is a variable format (a commonly used mapping), the multivalued DVA must be null before an entity can change roles. For example, suppose the Person class possesses a subclass named Employee that is mapped to a variable format of the PERSON data set. If the Person class were to contain a multivalued DVA named children that is mapped to an embedded data set, children would have to be null before a Person entity could become an Employee. Similarly, for an entity to be deleted from Employee but retained in Person, children would first have to be null.
- Embedded structures tend to waste more space than disjoint structures do.

These restrictions make the EMBEDDED-DATASET mapping option generally undesirable, which is the reason it is not a default mapping. However, the EMBEDDED-DATASET mapping option might be suitable in cases where performance better than that of the DISJOINT-DATASET mapping is needed and the OCCURRING-ITEM mapping is not viable (for example, in situations where a multivalued DVA cannot have a MAX limit or must be specified as DISTINCT).

### DATASET-OPTIONS

You can specify the DATASET-OPTIONS mapping option on a multivalued DVA that uses either the DISJOINT-DATASET mapping type or the EMBEDDED-DATASET mapping type. In both cases, the corresponding data set options can be used to control the TYPE and other physical characteristics of the data set. The syntax of the <data set options> parameter is described under “Data Set Options for a Base Class” earlier in this section. Although the semantics of the <data set options> parameter are generally the same for a multivalued DVA as they are for a class, there are some additional criteria to consider for the TYPE option with respect to a multivalued DVA.

These additional criteria are the following:

- When the multivalued DVA uses the EMBEDDED-DATASET mapping type, the TYPE of the data set cannot be DIRECT or RANDOM. Furthermore, an embedded STANDARD or COMPACT data set requires a spanning set; therefore, the multivalued DVA must be specified as DISTINCT to use these data set types.
- When the multivalued DVA uses a disjoint DIRECT or RANDOM data set, exactly one spanning set must be mapped to an appropriate access. Consequently, to meet this requirement, you must use the appropriate SET-OPTIONS syntax on the multivalued DVA or on a user-declared index that has the multivalued DVA as its key. (The SET-OPTIONS mapping option for multivalued DVAs is discussed under the heading “SET-OPTIONS” later in this section.)

Besides the TYPE option, other data set options can be specified to control the population, block size, family, and other physical characteristics of the data set.

For example, consider the following schema:

```

CLASS Person
  (children:  STRING [15], MV (MAX 10);
  );

DMSII-OPTIONS
  (CLASS Person
    (ATTRIBUTE children
      (MAPPING      = DISJOINT-DATASET;
      DATASET-OPTIONS
        (TYPE      = UNORDERED;
        POPULATION = 100000;
        BLOCKSIZE  = 500
        )
      )
    )
  );

```

This schema forces the multivalued DVA named children to be mapped to a disjoint data set even though an occurring item would have been the default. Furthermore, the disjoint data is to be unordered with a population of 100,000 records and a block size of 500 records.

### SET-OPTIONS

You can specify the SET-OPTIONS mapping option on a multivalued DVA whose mapping uses at least one set. The number and type of sets present on behalf of a multivalued DVA depends on several factors, as shown in Table 9–2.

Table 9-2. Factors Determining the Sets Present for a Multivalued DVA

| Mapping Type     | DISTINCT | UNIQUE | User Index Present | Sets Present                |
|------------------|----------|--------|--------------------|-----------------------------|
| DISJOINT-DATASET | —        | No     | No                 | SURROGATE-SET               |
|                  | No       | Yes    | No                 | SURROGATE-SET and VALUE-SET |
|                  | —        | —      | Yes                | SURROGATE-SET and VALUE-SET |
| OCCURRING-ITEM   | No       | No     | No                 | None                        |
| EMBEDDED-DATASET | No       | No     | No                 | None                        |
|                  | Yes      | No     | No                 | VALUE-SET                   |

As shown in the table, a multivalued DVA can have no sets, one set, or two sets, depending on its mapping type and on the following factors:

- Whether or not it is specified as **DISTINCT** or **UNIQUE**
- Whether or not it is a key of a user-declared index

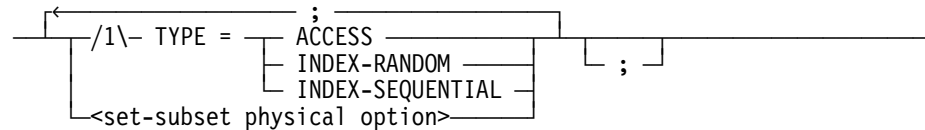
You can specify the **SET-OPTIONS** mapping option for any or all sets present, which means that a set option can be specified more than once for a given multivalued DVA. When you specify set options, you should use the keyword **SURROGATE-SET** or **VALUE-SET** to indicate the set to which the options pertain. If you use neither keyword, **VALUE-SET** is assumed by default.

Note that a user-declared index is allowed only on a multivalued DVA that is mapped to a disjoint data set, and the index is always the value set. In this situation, physical mapping options on the index are redundant with any **VALUE-SET** options present. If both are specified, the mapping options on the index take precedence.

### Syntax

When you specify the SET-OPTIONS mapping option, you can specify any of the available set options as indicated in the following diagram:

#### <set options>



As shown in the diagram, you can use set options to control the type and other physical characteristics of the corresponding set. You can use the TYPE option to document the default set type (INDEX-SEQUENTIAL) or to specify either of the alternate set types: INDEX-RANDOM or ACCESS. The INDEX-RANDOM type can be used for better performance if appropriate POPULATION and MODULUS options are specified. The ACCESS type can be used on a spanning set only when the data set type for the multivalued DVA is specified as DIRECT or RANDOM, thereby requiring an ACCESS set type.

The syntax and semantics of the <set-subset physical option> parameter are defined in the *DMSII DASDL Reference Manual*. Any available DMSII option can be used to specify such characteristics as TABLESIZE and MODULUS. The criteria for using these options to optimize a set are the same as for DMSII.

Specifying a data set as either DIRECT or RANDOM can provide a significant performance improvement for a multivalued DVA that meets the necessary criteria. For example, consider the following schema:

```

CLASS Person
  (license-numbers:  STRING [10], MV, UNIQUE;
   );

DMSII-OPTIONS
  (CLASS Person
    (ATTRIBUTE license-numbers
      (DATASET-OPTIONS
        (TYPE      = RANDOM;
         POPULATION = 100000;
        );
      SET-OPTIONS VALUE-SET
        (TYPE      = ACCESS;
         MODULUS   = 461;
        )
      )
    )
  );
  
```

The multivalued DVA license-numbers is mapped to the following DMSII data set:

```
LICENSE-NUMBERS RANDOM DATA SET
  (SIM-SURR-1      FIELD (48);
   DATA-LICENSE-NUMBE ALPHA (10);
  )
  POPULATION = 100000 RECORDS;

SIM-INDEX-1 SET OF LICENSE-NUMBERS
  KEY IS (SIM-SURR-1), DUPLICATES;

SIM-INDEX-2 ACCESS TO LICENSE-NUMBERS
  KEY IS (DATA-LICENSE-NUMBE), NO DUPLICATES,
  MODULUS = 461;
```

As shown, the DVA license-numbers is mapped to a disjoint data set of the type RANDOM. The value set is mapped to the ACCESS parameter of the data set, and POPULATION and MODULUS values have been chosen for efficient performance. If the DVA license-numbers were an unsigned, unscaled number with relatively dense values, a data set of the type DIRECT could be used, resulting in an even more efficient arrangement.

## EVA Mapping

A pair of EVAs that are inverses of each other form a single, bidirectional relationship. Even when an EVA is declared without an inverse, it still forms a bidirectional relationship because its inverse EVA is implicitly generated. Each EVA in a relationship possesses its own mapping information. However, when mapping each EVA, SIM considers the relationship as a whole rather than each EVA separately to ensure compatibility between the mapping of each EVA.

SIM offers two basic techniques (mapping types) for mapping an EVA, and both EVAs in a relationship are always assigned the same mapping type. These two mapping types are the following:

- Foreign-key mapping
- Common EVA data set mapping

The default mapping and the available mapping options used for EVAs are described in the following paragraphs.



## Default EVA Mapping

The default mapping type for a specific relationship depends on the relationship type as shown in the following table:

| Relationship Type | Default Mapping     |
|-------------------|---------------------|
| One-to-one        | Foreign-key         |
| One-to-many       | Common EVA data set |
| Many-to-many      | Common EVA data set |

Mapping options are available to specify a mapping type for an EVA and, in one case, to specify physical characteristics of the DMSII set generated on behalf of an EVA. These mapping options are described briefly under the next two headings:

- Single-Valued EVA Mapping Options
- Multivalued EVA Mapping Options

Subsequently, the semantics of available mapping options and information on when you might want to use them are given under the following headings:

- One-to-One Relationship Mapping
- One-to-Many Relationship Mapping
- Many-to-Many Relationship Mapping

## Single-Valued EVA Mapping Options

Information on specifying single-valued EVA mapping options is presented in the following paragraphs.

### Syntax

The following diagram shows the syntax to use for specifying a single-valued EVA mapping option:

**<single-valued EVA option>**

```

— MAPPING = —┐ FOREIGN-KEY —┐
                └ COMMON-DATASET ┘
  
```

### Explanation

The syntax elements shown in the diagram are explained briefly in the following table and are explained further in the text that succeeds the table.

| Syntax Element           | Explanation  |
|--------------------------|--|
| MAPPING = FOREIGN-KEY    | Specifies that the single-valued EVA is to be mapped to a foreign key.         |
| MAPPING = COMMON-DATASET | Specifies that the single-valued EVA is to be mapped to a common EVA data set. |

A single-valued EVA participates in either a one-to-one or a one-to-many relationship, depending on whether its inverse EVA is single valued or multivalued, respectively. FOREIGN-KEY is the default mapping type for a single-valued EVA that participates in a one-to-one relationship. COMMON-DATASET is the default mapping type for a single-valued EVA that participates in a one-to-many relationship.

You can use the single-valued EVA option to specify the default mapping type for documentation purposes, or you can use it to select a mapping type other than the default. Note that if you use the option for both EVAs in a relationship, it must be the same for both EVAs.

More semantics of the mapping types of a single-valued EVA are presented under the headings “One-to-One Relationship Mapping” and “One-to-Many Relationship Mapping” later in this section.

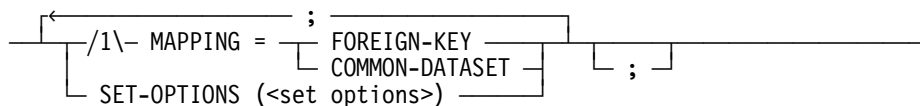
## Multivalued EVA Mapping Options

Information on specifying multivalued EVA mapping options is presented in the following paragraphs.

### Syntax

The following diagram shows the syntax to use for specifying a multivalued EVA option:

**<multivalued EVA option>**



**Explanation**

The syntax elements shown in the diagram are explained briefly in the following table and are explained further in the text that succeeds the table:

| Syntax Element              | Explanation  |
|-----------------------------|--|
| MAPPING = FOREIGN-KEY       | Specifies that the multivalued EVA is to be mapped to a foreign key.                                   |
| MAPPING = COMMON-DATASET    | Specifies that the multivalued EVA is to be mapped to a common EVA data set.                           |
| SET-OPTIONS (<set options>) | Specifies physical options for the set of the multivalued EVA when the EVA is mapped to a foreign key. |

A multivalued EVA participates in either a one-to-many or a many-to-many relationship, depending on whether its inverse EVA is single valued or multivalued, respectively. In both cases, COMMON-DATASET is the default mapping type. FOREIGN-KEY is an alternative mapping type for a multivalued EVA that participates in a one-to-many relationship. However, COMMON-DATASET is the only mapping type allowed for multivalued EVAs that participate in a many-to-many relationship.

You can use the multivalued EVA option to specify the default mapping type for documentation purposes, or you can use it to select a mapping type other than the default for a multivalued EVA that participates in a one-to-many relationship. Note that if you use the option for both EVAs in a relationship, it must be the same for both EVAs.

When a multivalued EVA is mapped to a foreign key, a DMSII set is generated on behalf of the EVA. By default, this set is specified as INDEX SEQUENTIAL with otherwise default DMSII mapping options. By using the SET-OPTIONS syntax, you can specify nondefault options such as TABLESIZE and LOADFACTOR.

More semantics of the MAPPING options and the SET-OPTIONS mapping option of a multivalued EVA are presented under the headings “One-to-Many Relationship Mapping” and “Many-to-Many Relationship Mapping.”

## One-to-One Relationship Mapping

You can map one-to-one EVA relationships with the FOREIGN-KEY mapping type (the default) or the COMMON-DATASET mapping type (an alternative). Each of these mapping types is described in the following paragraphs.

### Foreign-Key Mapping

When the single-valued EVAs of a one-to-one relationship are mapped with the FOREIGN-KEY mapping type, each of the data sets of their owner receives an item (the foreign key) that holds surrogate values for the appropriate target entities. To traverse an EVA, SIM uses the surrogate set of the target class for the EVA.

For example, consider the following schema:

```
CLASS Employee
  (mentor:      Manager, INVERSE IS apprentice;
  );

CLASS Manager
  (apprentice:  Employee, INVERSE IS mentor;
  );
```

In this example, the mentor and apprentice EVAs form a one-to-one relationship that is mapped to the following DASDL syntax:

```
EMPLOYEE DATA SET
  (SIM-SURR-1  FIELD (48);  % surrogate
  MENTOR      FIELD (48);  % foreign key to Manager
  );

SIM-INDEX-1 SET OF EMPLOYEE
  KEY IS (SIM-SURR-1), NO DUPLICATES;

MANAGER DATA SET
  (SIM-SURR-1  FIELD (48);  % surrogate
  APPRENTICE  FIELD (48);  % foreign key to Employee
  );

SIM-INDEX-2 SET OF MANAGER
  KEY IS (SIM-SURR-1), NO DUPLICATES;
```

Each data set has a SIM-SURR-1 item that is the surrogate for the corresponding class. The EVA apprentice and its inverse EVA, mentor, each has a foreign key in the data set of its owner. To traverse the EVA mentor, its foreign-key item value is extracted, and SIM-INDEX-2 is searched with that value. To traverse the EVA apprentice, its foreign-key item value is extracted, and SIM-INDEX-1 is searched with that value.

Foreign-key mapping is the best available technique for one-to-one relationships. EVA traversal typically requires only a few I/Os. Since each base class or subclass data set always has a surrogate set, no extra sets are needed to implement a one-to-one foreign-key relationship. Therefore, you can add an arbitrary number of one-to-one relationships without concern for approaching DMSII structure limits.

### Common Data Set Mapping

As an option, you can map one-to-one relationships using the COMMON-DATASET mapping type. This mapping type causes a relationship to be mapped to a specialized data set called the common EVA data set. This data set can have an arbitrary number of relationships mapped to it while requiring only a fixed number of structures. The common EVA data set mapping technique is described under the next heading, “One-to-Many Relationship Mapping.”

One-to-one relationships can be mapped using foreign keys without requiring extra structures. Moreover, the foreign-key technique is far more efficient for one-to-one relationships than is the common EVA data set technique. Therefore, common EVA data set mapping is not generally recommended for one-to-one relationships.

There is, however, one advantage of mapping a one-to-one relationship to the common EVA data set. Adding a new one-to-one relationship to an existing database by using the FOREIGN-KEY mapping type normally requires a database reorganization. Depending on the amount of data present, this process can take a considerable length of time. However, a new one-to-one relationship mapped to a common EVA data set can be added without requiring a reorganization. Therefore, this type of schema change can be very quick, depending on what other changes are made at the same time. Keep in mind, however, that, as indicated in Section 12, "Changing a SIM Database Schema," an EVA mapping option cannot change once the EVA is implemented.

## One-to-Many Relationship Mapping

You can map one-to-many relationships with the FOREIGN-KEY mapping type (an option) or the COMMON-DATASET mapping type (the default). Each of these mapping types is described in the following paragraphs.

### Foreign-Key Mapping

When a one-to-many relationship is mapped with nondefault foreign-key mapping, the multivalued EVA is mapped to a DMSII set that spans the data set of the multivalued EVA target. The key of the DMSII set is the foreign-key item of the single-valued EVA.

For example, consider the following schema:

```
CLASS Employee
  (adviser:    Manager, INVERSE IS advisees;
   );

CLASS Manager
  (advisees:   Employee, MV, INVERSE IS adviser;
   );

DMSII-OPTIONS
  (CLASS Employee (ATTRIBUTE adviser (MAPPING = FOREIGN-KEY))
   );
```

In this example, the EVAs advisees and adviser form a one-to-many relationship. This relationship is explicitly mapped by means of a DMSII mapping-options declaration that uses the FOREIGN-KEY option. Note that only adviser is explicitly mapped by using the FOREIGN-KEY option. SIM automatically chooses the same mapping type for the inverse EVA, advisees. This schema results in the following DMSII DASDL syntax:

```
EMPLOYEE DATA SET
  (SIM-SURR-1  FIELD (48);  % surrogate
   ADVISER     FIELD (48)   % foreign key to Manager
  );

SIM-INDEX-1 SET OF EMPLOYEE
  KEY IS (SIM-SURR-1), NO DUPLICATES;

ADVISEES SET OF EMPLOYEE
  KEY IS (ADVISER), DUPLICATES;

MANAGER DATA SET
  (SIM-SURR-1  FIELD (48);  % surrogate
  );

SIM-INDEX-2 SET OF MANAGER
  KEY IS (SIM-SURR-1), NO DUPLICATES;
```

As in the case of one-to-one relationship mapping, the single-valued EVA adviser causes a foreign-key item to be placed in the data set EMPLOYEE of its owner. To traverse the EVA adviser, its foreign-key item value is extracted, and SIM-INDEX-2 is searched with that value. To traverse the multivalued EVA advisees, the surrogate value for the Manager class is extracted (from SIM-SURR-1 of MANAGER), and the set ADVISEES is searched for all instances with that surrogate value.

The FOREIGN-KEY mapping type requires the addition of the extra set on behalf of the multivalued EVA. Since one-to-many relationships are the most common relationship type in most applications, a large application could require a significant number of structures to implement all one-to-many relations with the foreign-key technique. The extra structures require additional memory and increase the chance of reaching the DMSII structure limit. For these reasons, SIM does not use the FOREIGN-KEY mapping type for one-to-many relationships by default.

However, the foreign-key technique is much more efficient than the default common EVA data set technique because EVA traversal and update is significantly faster with the foreign-key mapping. Therefore, if performance is important or if your application is not very large, it is strongly recommended that you select the FOREIGN-KEY mapping type for each one-to-many relationship by specifying *MAPPING = FOREIGN-KEY* for either EVA involved in the relationship.

When a one-to-many relationship is mapped with the foreign-key technique, you also can specify the SET-OPTIONS mapping option for the multivalued EVA to optimize the set that is generated on its behalf. By using an INDEX-RANDOM structure type or by setting such options as TABLESIZE and LOADFACTOR, you might be able to improve the performance of the set over its default specifications.

## Common Data Set Mapping

By default, one-to-many relationships are mapped to a common EVA data set. This mapping technique causes the relationship to be mapped to a standard data set that is spanned by two INDEX SEQUENTIAL sets.

For example, if the adviser/advisees relationship in the previous example were mapped using the common EVA data set technique, the following DMSII DASDL syntax would result instead:

```

EMPLOYEE DATA SET
  (SIM-SURR-1      FIELD (48);   % surrogate
  );

SIM-INDEX-1 SET OF EMPLOYEE
  KEY IS (SIM-SURR-1), NO DUPLICATES;

MANAGER DATA SET
  (SIM-SURR-1      FIELD (48);   % surrogate
  );

SIM-INDEX-2 SET OF MANAGER
  KEY IS (SIM-SURR-1), NO DUPLICATES;

SIM-EVAS DATA SET
  (SIM-EVA-1      FIELD (48);   % primary EVA item
  SIM-REL-NUMBER  FIELD (48);
  SIM-EVA-2      FIELD (48);   % secondary EVA item
  );

SIM-EVASET-1 SET OF SIM-EVAS      % secondary EVA set
  KEY IS (SIM-EVA-1, SIM-REL-NUMBER, SIM-EVA-2), NO DUPLICATES;

SIM-EVASET-2 SET OF SIM-EVAS      % primary EVA set
  KEY IS (SIM-EVA-2, SIM-REL-NUMBER, SIM-EVA-1), NO DUPLICATES;

```

Each record in the common EVA data set, SIM-EVAS, holds one relationship instance. The SIM-EVA-1 and SIM-EVA-2 items hold the surrogate values of the two entities involved in the relationship instance. Therefore, the type for these items is always compatible with the surrogate item in the corresponding data sets. The SIM-REL-NUMBER item holds a relationship number that uniquely identifies each relationship in the database.

The SIM-EVASET-1 and SIM-EVASET-2 sets, which span the common EVA data set, allow relationships to be traversed with the surrogate value of either entity. Since these sets specify NO DUPLICATES, this common EVA data set can handle one-to-one relationships, one-to-many relationships, and many-to-many relationships that do not allow duplicates (that is, DISTINCT relationships).

Any number of additional relationships can be mapped to the same common EVA data set, provided that the following conditions are met:

- The surrogate items of the two classes involved in the relationship are compatible with the SIM-EVA-1 and SIM-EVA-2 items used by the common EVA data set.
- The relationship type is compatible with the DUPLICATES specification used by the spanning sets for the data set.

When a new relationship is specified with (or defaults to) *MAPPING = COMMON-DATASET* and no compatible common EVA data set exists, a suitable common EVA data set is generated, together with two new sets. Therefore, it is possible to have several common EVA data sets within a single database.

The advantage of the common EVA data set mapping is that an arbitrarily large number of relationships can be mapped to a single data set, thereby preventing a proliferation of new DMSII structures. However, as indicated earlier under “Foreign-Key Mapping,” foreign-key mapping is much more efficient for one-to-many relationships.

When a multivalued EVA is mapped with the COMMON-DATASET mapping type, it cannot have the SET-OPTIONS mapping option defined for it.

### Many-to-Many Relationship Mapping

Both multivalued EVAs of all many-to-many relationships are mapped with the common EVA data set mapping type described under the preceding heading, “Common Data Set Mapping.”

Many-to-many relationships that do not allow duplicates (namely, those in which either or both EVAs specify the DISTINCT option) are mapped to a common EVA data set whose spanning sets are declared with the NO DUPLICATES option. Such a data set can be shared by one-to-one and one-to-many relationships as well.

Many-to-many relationships that allow duplicates (namely, those in which neither EVA specifies the DISTINCT option) are mapped to a common EVA data set whose spanning sets specify the DUPLICATES option.

No alternative mapping techniques are offered for many-to-many relationships, and no mapping options are offered for the common EVA data set or its sets. However, when the owning class of each EVA uses the default (TIMESTAMP) surrogate type, SIM automatically optimizes certain aspects of the common EVA data set and its sets. These optimizations ensure minimal table levels within the spanning sets and ensure a large number of relationship instances.



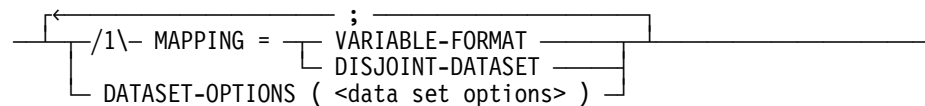
## Subclass Mapping Options

A subclass has several mapping options that affect the way it is mapped into DMSII constructs.

### Syntax

The following diagram shows the syntax to use for specifying subclass mapping options:

**<subclass options>**



### Explanation

The syntax elements shown in the diagram are explained briefly in the following table and are explained in more detail in the paragraphs that succeed the table.

| Syntax Element                       | Explanation   |
|--------------------------------------|---|
| MAPPING = VARIABLE-FORMAT            | Specifies that the subclass is to be mapped to a variable format record of the data set of its owner. |
| MAPPING = DISJOINT-DATASET           | Specifies that the subclass is to be mapped to a disjoint data set.                                   |
| DATASET-OPTIONS (<data set options>) | Specifies physical options for the disjoint data set to which the subclass is mapped.                 |

The most basic physical option for a subclass is its MAPPING option. Depending on what you specify in the MAPPING option, you can also specify the DATASET-OPTIONS mapping option. These two kinds of subclass mapping options are described in the following paragraphs.

## MAPPING Option

A subclass can be mapped to one of the following:

- A variable format of the data set of its owner
- A new disjoint data set

SIM chooses a default mapping for a subclass based on certain characteristics of the subclass. You might be able to achieve better performance in some instances by using the MAPPING option to change the default mapping for a subclass. A MAPPING option can be one of the following types:

- VARIABLE-FORMAT option
- DISJOINT-DATASET option

### VARIABLE-FORMAT Option

The VARIABLE-FORMAT option is allowed only for a subclass that meets all of the following criteria:

- The subclass has a single superclass.
- The subrole DVA that defines the subclass is single valued (SV).
- The superclass is mapped to a standard or unordered data set.

The VARIABLE-FORMAT option is chosen by SIM whenever possible, whether or not you specify it in the schema file. This option causes the subclass to be mapped to a variable record format of the data set to which its superclass is mapped. Use of the VARIABLE-FORMAT option is shown in the following example.

#### Example

Consider the following schema:

```
CLASS Employee
  (emp-no:      NUMBER [5], REQUIRED, UNIQUE;
   employee-role: SUBROLE (Manager, Laborer);
  );

SUBCLASS Manager OF Employee
  (salary:      NUMBER [10,2], REQUIRED
  );

SUBCLASS Laborer OF Employee
  (hourly-rate: NUMBER [10,2]
  );
```

The subclasses Manager and Laborer meet the required criteria. Therefore, they are mapped to a variable record format of the data set for the Employee class. The implicit DMSII-OPTIONS specification for the subclasses is the following:

```
DMSII-OPTIONS
  (SUBCLASS Manager (MAPPING = VARIABLE-FORMAT);
   SUBCLASS Laborer (MAPPING = VARIABLE-FORMAT);
  );
```

Whether or not the MAPPING options for the two subclasses are explicitly stated in the schema file, the resulting DASDL syntax appears as follows:

```

EMPLOYEE DATASET
  (SIM-SURR-1  FIELD (48) REQUIRED;
   EMP-NO     NUMBER (5);
   SIM-RECTYPE RECORD TYPE (254);
  )
1: (          % variable-format for Manager
   SALARY     NUMBER (10,2);
2: (          % variable-format for Laborer
   HOURLY-RATE NUMBER (10,2);
  );

SIM-SET-1 SET OF EMPLOYEE
  KEY IS (SIM-SURR-1), NO DUPLICATES;

```

Note that the Employee data set was defined as a variable record format. Note also that the attributes of Manager and Laborer were mapped to separate format numbers.

Advantages of the VARIABLE-FORMAT option are the following:

- It allows many subclasses to be mapped to a single data set. Fewer structures are needed to map the subclasses to a single data set than are needed to map each of the subclasses to a separate data set.
- It allows each of the subclasses mapped to the same data set to share the same surrogate item and surrogate set, thereby further reducing the number of structures needed.

Some disadvantages of the VARIABLE-FORMAT option are the following:

- If too many subclasses with a large number of attributes are mapped to the same data set, the data set might acquire more items than DMSII permits. This excess of items would cause the DASDL compiler to indicate a PROPERTY ARRAY SIZE EXCEEDED error.
- Linear searches on a variable-format, mapped subclass are slower. For example, if the Employee class contains 10,000 entities and the Manager subclass includes only 500 of them, a linear search on Manager still requires the examination of all 10,000 records in the data set.
- When a new subclass is added to an existing data set or when an existing subclass is deleted, a database reorganization is required to effect the change. With MAPPING options other than the VARIABLE-FORMAT option, the reorganization might not be needed.

### DISJOINT-DATASET Option

The DISJOINT-DATASET option is the default MAPPING option for all subclasses that cannot be mapped with the VARIABLE-FORMAT option. The DISJOINT-DATASET option causes the subclass to be mapped to a new disjoint data set containing a new surrogate item and a surrogate set.

#### Example

For example, assume that the employee-roles subrole DVA in the preceding example was multivalued (MV). SIM would then assume the following DMSII-OPTIONS values even if you did not explicitly state them in the schema file:

```
DMSII-OPTIONS
  (SUBCLASS Manager (MAPPING = DISJOINT-DATASET);
   SUBCLASS Laborer (MAPPING = DISJOINT-DATASET);
  );
```

The resulting DASDL syntax for Person, Manager, and Laborer would be the following:

```
EMPLOYEE DATASET
  (SIM-SURR-1  FIELD (48) REQUIRED;
   EMP-NO      NUMBER (5);
  );

SIM-SET-1 SET OF EMPLOYEE
  KEY IS (SIM-SURR-1), NO DUPLICATES;

MANAGER DATASET
  (SIM-SURR-1  FIELD (48) REQUIRED;
   SALARY      NUMBER (10,2);
  );

SIM-SET-2 SET OF MANAGER
  KEY IS (SIM-SURR-1), NO DUPLICATES;

LABORER DATASET
  (SIM-SURR-1  FIELD (48) REQUIRED;
   HOURLY-RATE NUMBER (10,2);
  );

SIM-SET-3 SET OF LABORER
  KEY IS (SIM-SURR-1), NO DUPLICATES;
```

You can specify the DISJOINT-DATASET option for any subclass, including those subclasses that would be mapped through the VARIABLE-FORMAT option by default.

An obvious disadvantage of the DISJOINT-DATASET option is the use of additional structures, which contributes to increasing the memory, disk, and update time required by the database. However, this disadvantage is circumvented by the following advantages of the DISJOINT-DATASET option:

- Each data set requires fewer items. Therefore, the risk of exceeding DMSII item limits is reduced.
- Linear searches access only the minimum possible number of records. Therefore, the time required for these operations is reduced.
- Since a reorganization is not needed to add or delete a data set and its accompanying sets, adding or deleting a subclass through the DISJOINT-DATASET option might eliminate the need for a reorganization.

The DISJOINT-DATASET option is recommended for mapping subclasses when any of these advantages are desired.

### Data Set Options for a Subclass

When a subclass is mapped to a disjoint data set, either by default or through the specification of a DISJOINT-DATASET option in the database schema file, you can specify data set options for the subclass.

As is the case for base class DATASET-OPTIONS mapping options, a recommended use of subclass DATASET-OPTIONS mapping options is to set the expected population limit of the subclass. Other kinds of options can be used to optimize the data set type, the space usage, or the performance aspects of the data set to which the subclass is mapped.

### Example

For example, consider the following portion of schema:

```
SUBCLASS Manager OF Employee
  (salary:      NUMBER [10,2], REQUIRED
  );

INDEX Manager-access ON Manager
  (salary);

DMSII-OPTIONS
  (SUBCLASS Manager
    (MAPPING = DISJOINT-DATASET;
    DATASET-OPTIONS
      (TYPE      = RANDOM;
      POPULATION = 1000;
      );
    );
  INDEX Manager-access
    (SET-OPTIONS
      (TYPE      = ACCESS;
      MODULUS    = 125;
      );
    );
  );
```

The set of options in this schema causes the Manager subclass to be mapped to a disjoint random data set with a population of 1000 records. The ACCESS key for the data set is salary with a modulus of 125. The DASDL syntax that results from these options is the following:

```
MANAGER RANDOM DATASET
  (SALARY  NUMBER (10,2);
  );

MANAGER-ACCESS ACCESS TO MANAGER
  MODULUS = 125
  KEY IS (SALARY);
```

To achieve better performance, the specification of DATASET-OPTIONS mapping options is recommended whenever the access characteristics of a subclass are known.



### DUPLICATES Option

All DMSII sets, subsets, and accesses possess a DUPLICATES option that defines the following:

- Whether duplicate entries are allowed
- Whether the key can change if duplicates are not allowed
- The order in which to store the duplicates if they are allowed

When mapping an index, SIM chooses a default DUPLICATES option that is compatible with the UNIQUE option of the index and provides a balance between performance and flexibility.

You can specify a DUPLICATES option to reflect the default for documentation purposes, or you can choose a nondefault DUPLICATES option for performance reasons. However, any DUPLICATES option that you specify for an index must be compatible with the UNIQUE option of the index.

An index can use the DUPLICATES option in addition to or instead of the SET-OPTIONS option.

If an index is declared with the UNIQUE option or if any of its key attributes are set with the UNIQUE option, then the index can use only the DUPLICATES options KEYCHANGEOK or NONE. These options are explained in the following table and in the paragraph that follows the table.

| Option      | Explanation  |
|-------------|--|
| KEYCHANGEOK | This option is the default option for indexes that are specified as UNIQUE. It is allowed only when the index is mapped to a set or a subset. It causes the corresponding structure to be generated with the NO DUPLICATES KEYCHANGEOK option. The KEYCHANGEOK option allows the key attribute or attributes of the index to change values. However, it has a slight performance overhead owing to the extra checking that must be performed by DMSII. |
| NONE        | This option causes the corresponding set, subset, or access of the index to be declared with the NO DUPLICATES option. Once the key attribute or attributes are assigned a value, the attribute values cannot be changed. The NONE option has the best performance characteristic for indexes specified as UNIQUE. However, the option has the disadvantage that the key attributes of the index cannot be changed.                                    |



One exception to the rules governing the **KEYCHANGEOK** and **NONE** options is for a user-declared index that is used as a surrogate index. This exception occurs when the target structure of the index is a base class that uses the attribute **SURROGATE** option and the only key attribute of the index is the surrogate attribute. In this situation, the index is used as the surrogate index, and its mapped set or access cannot use the **KEYCHANGEOK** option. Therefore, if you specify a **DUPLICATES** option for a surrogate index, it must be the **NONE** option.

If neither the index nor any of its key attributes are specified as **UNIQUE**, the index can only use the **DUPLICATES** options **ALLOWED**, **FIRST**, or **LAST**. These options are explained in the following table:

| Option  | Explanation  |
|---------|--|
| ALLOWED | <p>This is the default option for indexes that are not <b>UNIQUE</b>. It is allowed when the index is mapped to a set or a subset or to the access of a random data set. It causes the corresponding structure to be generated with the <b>DUPLICATES</b> option, which results in a “duplicates resolver” word being added to each key entry.</p> <p>Although specifying the <b>ALLOWED</b> option results in larger key and table sizes, it causes the performance of the structure to be improved during deletion operations.</p> |
| FIRST   | <p>This option is allowed only when the index is mapped to a set or a subset. It causes the corresponding structure to be generated with the <b>DUPLICATES FIRST</b> option, whereby duplicate key entries are stored in most-recent-first order.</p>  |
| LAST    | <p>This option is allowed only when the index is mapped to a set or a subset. It causes the corresponding structure to be generated with the <b>DUPLICATES LAST</b> option, whereby duplicate key entries are stored in most-recent-last order.</p>  |

## SET-OPTIONS Option for an Index

If the target structure of an index is mapped to a disjoint data set and the first key attribute of the index is specified as **REQUIRED**, the index is mapped to a set. If the target structure of the index is variable-format or if any of the keys of the index are not specified as **REQUIRED**, the index is mapped to an automatic subset. In either case, **SIM** chooses an **INDEX SEQUENTIAL** structure type by default.

By specifying the **TYPE** option, you can choose a structure type other than **INDEX-SEQUENTIAL**, such as **INDEX-RANDOM**. If an index is normally mapped to a set, you can use the **TYPE** option to map the index to an access rather than to a set. However, you cannot use the **ACCESS** option when the index would normally be mapped to a subset. Furthermore, when the **ACCESS** option is used, the target class or subclass of the index must use data set options that specify a data set type that requires an access, such as a **RANDOM** or **DIRECT** data set type.

### Example

For example, consider the following schema:

```
CLASS Employee
  (emp-no:  NUMBER [5], REQUIRED, UNIQUE;
  );

INDEX Employee-numbers ON Employee
  (emp-no) UNIQUE;

DMSII-OPTIONS
  (CLASS Employee (DATASET-OPTIONS (TYPE = DIRECT)));
  INDEX Employee-numbers (SET-OPTIONS (TYPE = ACCESS));
  );
```

Because the Employee class is mapped to a disjoint data set and the emp-no attribute is specified as REQUIRED, the Employee-numbers index would normally be mapped to an INDEX SEQUENTIAL set. This fact, together with the fact that the data set for Employee has been specified as DIRECT, allows the index to be mapped to an access. If emp-no were not specified as REQUIRED, Employee-numbers would be mapped to a subset; consequently, its mapping could not be changed to an access.

Note that a data set specified as RANDOM or DIRECT must have an access declared for it.

Also note that the key of a direct data set must be a single, unsigned, unscaled NUMBER item with 12 or fewer digits of precision.

When an index is mapped to the access of a random data set, you can specify a modulus for the index as an option.

When an index is not mapped to an access, you can specify physical options within the SET-OPTIONS option to optimize space and performance characteristics. The most important of these physical options is TABLESIZE. Since the same criteria are used for optimizing a set or a subset in SIM and in DMSII, the same evaluation criteria should be used. In particular, the TABLESIZE option should be set, as well as the POPULATION option of the target data set for the index, to minimize the number of levels in the set or the subset. Minimizing table levels reduces the number of I/Os required and reduces the processor overhead during update and retrieval operations.

**Example**

For example, consider the following schema:

```
CLASS Employee
  (name:  STRING [30]
  );

INDEX Employee-by-name ON Employee
  (name);

DMSII-OPTIONS
  (INDEX Employee-by-name
    (SET-OPTIONS
      (TABLESIZE = 128;
        LOADFACTOR = 85;
      )
    )
  );
```

In this example, two physical options are set for the mapped subset of the index Employee-by-name. The TABLESIZE option is set to 128 entries per table, and the LOADFACTOR option is set to 85 percent. Since the POPULATION option of the data set defaults to 10,000 records, the given set options ensure that the subset requires only two levels of tables. Without these set options, DMSII would have chosen a much smaller TABLESIZE (approximately 33 entries per table) and a sparser LOADFACTOR (66 percent). For a population of 10,000 records, these defaults would have required the subset to use three table levels, resulting in a less efficient implementation.



## Section 10

### Dictionary Options

A dictionary-options declaration allows you to integrate a SIM database schema that is generated through the SIM Utility interface or the SIM Design Assistant (SDA) with the Advanced Data Dictionary System (ADDs). This section describes the use of ODL to specify a dictionary-options declaration and presents information on using the declaration. All examples presented in this section reflect the ORGANIZATION database, which is described in Section 3, "Basic ODL Considerations."

## Syntax

The following diagram shows the syntax to use for specifying a dictionary-options declaration:

**<dictionary-options declaration>**

```

— DICTIONARY-OPTIONS ( ————— ; ————— )
                        [ /1*\- DICTIONARY = <dictionary name> ]
                        [ /1*\- DIRECTORY = <directory name> ]
→ [ ; ] ) —————

```

### Explanation

The syntax elements shown in the diagram are explained in the following table:

| Syntax Element                 | Explanation   |
|--------------------------------|---|
| DICTIONARY-OPTIONS             | Identifies the declaration to SIM as a dictionary-options declaration.  |
| DICTIONARY = <dictionary name> | Indicates the name of the ADDS database with which the SIM Utility is to integrate the generated SIM database schema. This name is the DMSUPPORT library function name (for example, DATADictionary). |
| DIRECTORY = <directory name>   | Indicates the name of the ADDS directory in which the generated SIM database schema is to be stored (for example, JONES).   |

Note that both a dictionary and a directory must be named in the declaration.

# Using a Dictionary-Options Declaration

Including a dictionary-options declaration in a SIM database schema is optional, with the following exception: If the option `REPOSITORY FOR SIM GENERATION = OPTIONAL` is not specified in the InfoExec SL/CONFIG file at your site and the schema validation or generation process is not initiated from ADDS, then you must specify a dictionary-options declaration.

If you do include a dictionary-options declaration, you specify it in the CARD file. You can use the declaration with the `ADD` and `CHANGE` commands of the SIM Utility. When the declaration is optional, you can validate or generate the schema before the schema is loaded into ADDS.

When a dictionary-options declaration is first processed, it causes the schema to be loaded into the indicated dictionary and directory, using the default status and security. The lowest available version number within the given directory is automatically used.

Once a schema has been generated or changed with a dictionary-options declaration, the database is marked as active with respect to ADDS. The specified dictionary and directory names are stored in the SIM directory of the database. Thereafter, all changes to the schema are automatically mapped back into ADDS using the same dictionary and directory names. In this way, ADDS is kept synchronized with the current description of the database.

Note that when ADDS initiates the SIM Utility, the dictionary-options declaration is not used. Instead, ADDS passes the equivalent information to the SIM Utility programmatically.

## Advantages

Some of the primary advantages of using the dictionary-options declaration to integrate a SIM database with ADDS are the following:

- ADDS security features can then be used to control certain SIM Utility functions on the database.
- ADDS tracking features can then be used to track programs that use the host language interface (HLI) to access the database.
- ADDS reporting features can then be used to generate various reports about the database, such as listings and cross-references.

## Effect on a Database

The effect of the dictionary-options declaration on a SIM database is further described in the following paragraphs. Refer to the *InfoExec ADDS Operations Guide* for details on using ADDS with a SIM database.

When the dictionary-options declaration is first used for a database through an ADD command or a CHANGE command, the schema is loaded into ADDS for the first time. To determine whether the schema can be loaded into ADDS, the SIM Utility performs the following checks:

- The specified dictionary must be available.
- The user who initiated the SIM Utility must have update access to the dictionary.
- The user must be able to create the specified directory if it does not exist. If the specified directory already exists, the user must be allowed to create new entities within it.
- The user must be allowed to create a SIM database entity with the database name used in the <DB title> construct given to the SIM Utility.

If these checks pass, a successful ADD/GENERATE or CHANGE/GENERATE run concludes with the SIM database schema being loaded into ADDS. From that point on, the database is active with respect to ADDS, and subsequent SIM Utility actions on the database invoke various security checks.

When a CHANGE, UPDATE, or REINITIALIZE command is issued for a SIM database that is active with respect to ADDS, the SIM Utility performs the following checks:

- The dictionary with which the database is active must be available.
- The user must have both inquiry and update access to the SIM database entity within ADDS.
- A user performing the CHANGE command must have access to the dictionary.

If any of these checks fail, the user is not allowed to perform the requested function, and an error is generated. Otherwise, the function can be performed.

After dictionary options have been initially specified for a database, they need not be specified again in a subsequent CHANGE command unless they are being modified. When a modified dictionary-options declaration is first specified in a CHANGE command, the SIM Utility performs the following checks:

- If the dictionary name is changed, the new dictionary must be present and the user must have update access to the dictionary.
- If the directory name is changed, the user must be able to create the new directory if it does not exist, or be able to perform updates within the directory if it does exist.
- The user must be allowed to create a SIM database entity using the database name.

If these checks pass, the SIM Utility establishes the new dictionary and/or directory names as the current description location of the database within ADDS.

Whenever a CHANGE command is performed, the SIM Utility loads the new schema into ADDS, using the latest dictionary and directory names and the default status and security. Existing descriptions using the same dictionary and/or directory names are not disturbed; the new description is always stored with the lowest available (next) version number.

When a LIST command is submitted for a SIM database that is active with respect to ADDS, the SIM Utility performs the following checks:

- The dictionary for the database must be available.
- The user submitting the LIST command must have inquiry access to the current database entity within ADDS.

If these checks pass, the LIST function is performed.

## Example of a Dictionary-Options Declaration

The following example shows portions of a SIM database schema that includes a dictionary-options declaration. This schema can be used with an ADD command or a CHANGE command.

```
CLASS Person
  (name      : STRING[20];
   birthdate : DATE;

   ...

);

DICTIONARY-OPTIONS
  (DICTIONARY = DATADictionary;
   DIRECTORY  = DBTEST;
  );
```

The dictionary-options declaration shown designates that the database schema be stored in an ADDS dictionary named DATADictionary and that it be stored under the directory DBTEST.



The description could subsequently be moved to a new directory by submitting the following schema with a **CHANGE** command:

```
CLASS Person
  (name      : STRING[30];    % Increased from 20
   birthdate : DATE;

   ...

);

DICTIONARY-OPTIONS
  (DICTIONARY = DATADictionary;
   DIRECTORY  = DBPROD;
  );
```

This amended schema would be stored in the same dictionary, but under a different directory, namely, **DBPROD**.



# Section 11

## SIM Utility

The SIM Utility provides an interface for creating and maintaining SIM or SQLDB databases. You can use the utility to perform the following functions by means of the commands noted in parentheses:

- Create a new database (ADD command).
- Change the schema of an existing database (CHANGE command).
- Update an existing database to a new level of software (UPDATE command).
- Reinitialize data in an existing database (REINITIALIZE command).
- List the schema of an existing database (LIST command).

Each of these functions is invoked by running the utility with the appropriate command specified as a parameter. Depending on the command, the utility allows or requires additional input through input files. As the utility performs a function, it produces output files and it might interact with a remote terminal.

This section describes the operation of the SIM Utility with respect to SIM databases. For information on the use of the SIM Utility for SQLDB databases, refer to the *InfoExec Structured Query Language Database (SQLDB) Programming Guide*.

The following topics are presented in this section:

- The types of SIM databases that can be created by the SIM Utility
- Basic considerations for executing the SIM Utility
- The SIM Utility commands for creating and maintaining a SIM database
- The SIM Utility files that can be file-equated to direct the utility to find or create specific files
- The SIM Utility options for process and output control

## Database Types

The SIM Utility can create the following types of SIM databases, each of which serves a different purpose:

- Validated
- Generated

- Viewed

These database types are described in the following paragraphs.

### Validated Databases

A validated database is created by the SIM Utility when it is given an ADD command or a CHANGE command with the VALIDATE option specified. Such a command is called a VALIDATE run, and it serves to verify the syntax and semantics of a database schema without affecting an existing generated database. A validated database consists of only a database directory file. During a VALIDATE run, the utility creates this file with the following name:

DESCRIPTION/<database name>/SIM-DIRECTORY

The database name is the name specified to the SIM Utility when the VALIDATE run is performed. The usercode and pack name for this file are always those specified for the control file (although a control file is not actually created). Refer to Section 9, “DMSII Mapping Options,” for more information on control file specifications.

A validated database can be used only for the following functions:

- A SIM host language interface (HLI) program can be compiled against a validated database. If a validated database exists, the HLI compilers use this database instead of using a generated database.
- The SIM Utility can perform a LIST command on a validated database to extract the corresponding database schema.
- An application program using the OML library interface can open a validated database for inquiry and perform queries on the metaschema classes of the directory.
- The schema can be accessed by the SIM Design Assistant (SDA) product.
- The schema can be loaded by the ADDS *LOADDB* function.

### Generated Databases

A generated database is created by the SIM Utility when it is given an ADD command or a CHANGE command with the GENERATE option specified. Such a command is called a GENERATE run, and it serves to verify the syntax and semantics of a database schema as well as to generate the corresponding database. A generated SIM database is an audited DMSII database consisting of the normal DESCRIPTION, DMSUPPORT, CONTROL, AUDIT, and structure files.

In a generated database, the directory is implemented as a data set named SIM-DIRECTORY. The directory is contained in a file with the following name:

<database name>/SIM-DIRECTORY/DATA

The usercode and pack name for this file are always those specified for the control file of the database. Refer to Section 9, “DMSII Mapping Options,” for more information on control file specifications.

All SIM Utility commands can be used on a generated database. SIM HLI programs can be compiled against a generated database. SIM application programs, including the InfoExec query products, can open a generated database for inquiry or update, and the programs can perform all normal OML operations. Also, the SIM Design Assistant (SDA) can access the database, and the ADDS LOADDDB function can be used on the database.

## Viewed Databases

A viewed database is created by the SIM Utility when it is invoked with either the LINC.View or DMS.View facility from ADDS. These facilities create a SIM view of an existing DMSII database by creating a validated directory file. The directory file for a viewed database has the same name, usercode, and pack name as those for a validated database.

The functions that can be performed on a viewed database are the following:

- A SIM HLI program can be compiled against a viewed database.
- The SIM Utility can perform a LIST command on a viewed database to extract the corresponding database schema.
- A SIM application program using either the HLI interface or the OML library interface can open a viewed database for inquiry and perform inquiry queries.
- A SIM application program can open an updatable viewed database for update and perform both inquiry and update queries. An updatable viewed database is a viewed database created by means of the DMS.View facility with *Y* specified for the option titled “Allow Update Queries?”. Refer to the *InfoExec DMS.View Operations Guide* for more information on the DMS.View facility.
- The schema can be accessed by the InfoExec query products
- The schema can be accessed by the SIM Design Assistant product.
- The schema can be loaded by the ADDS *LOADDDB* function.

Refer to the *InfoExec LINC.View Operations Guide* for information on the LINC.View facility.

# Executing the SIM Utility

Basic considerations for executing the SIM Utility are presented in the following paragraphs.

## Program Parameter

When the SIM Utility is run, it must be given a program parameter that defines a valid command. Descriptions of the commands that you can specify as the parameter are provided under the heading “Utility Commands” later in this section. The parameter must observe the following conventions:

- It can have arbitrary blanks provided that they are between tokens. That is, the parameter can be designated in free-form input.
- It can be designated in intermixed uppercase and lowercase letters, except that strings enclosed in quotation marks must use the precise case desired.
- It cannot contain any nondisplayable characters, except that one or more null characters can be placed at the end of the command.

## Input Files

Depending on the command, the SIM Utility can use additional input from various input files. The primary input file is the CARD file, which typically holds an input ODL database schema. The DASDL file is sometimes used to access a DMSII description file. The CARD and DASDL files can be file-equated to direct the SIM Utility to specific files. SIM Utility files are described under the heading “Utility Files” later in this section.

## Output Files

All SIM Utility commands produce some kind of output that consists of one or more files. The primary output file is the DDLRESULTS file, which contains a summary of the execution of the SIM Utility. In addition, the SIM Utility can produce a printer listing by means of the LINE file, and it can produce a new DMSII description file by means of the DASDL file. SIM Utility files are described under the heading “Utility Files” later in this section.

## Remote Execution

You can run the SIM Utility from a remote terminal, using any of the following commands:

- CANDE *RUN* or CANDE *EXECUTE* command
- CANDE *UTILITY* command
- MARC *RUN* command

If you use the CANDE *RUN* or CANDE *EXECUTE* command, you must specify the parameter as a quoted string enclosed in parentheses. An example of a CANDE *RUN* command is the following:

```
RUN *SYSTEM/SIM/UTILITY ON PACK ("ADD DATABASE Organization");
FILE CARD=SCHEMA/ORGANIZATION;
```

This command runs the SIM Utility, directs it to generate a new database named ORGANIZATION, and directs it to find the corresponding database schema in a file named SCHEMA/ORGANIZATION.

If you use the CANDE *UTILITY* command to run the utility, the need for quotation marks and parentheses is eliminated. The following example CANDE *UTILITY* command provides the same instruction as does the preceding command:

```
U *SYSTEM/SIM/UTILITY ON PACK ADD DATABASE Organization;
FILE CARD=SCHEMA/ORGANIZATION;
```

When the SIM Utility is run from a remote terminal, the default output option for the LIST command is REMOTE. The REMOTE option causes the output to be displayed, one page at a time, on the terminal from which the LIST command was executed. When the SIM Utility is run through a WFL job, the default output option for the LIST command is PRINTER. The PRINTER option causes a printer listing to be generated. The LIST command is described under the heading “Utility Commands” later in this section.

When the SIM Utility completes its run, it displays a message that summarizes its results. The message indicates if any errors or warnings were detected.

## Batch Execution

The SIM Utility can be run in batch mode by initiation through a WFL job. The parameter must be designated as a quoted string enclosed in parentheses. If the SIM Utility detects any errors, it returns the number of errors found as a negative task value. This value can be detected by the initiating WFL job. For example, consider the following WFL job:

```
BEGIN JOB GENERATE/ORGANIZATION;
TASK T;
RUN *SYSTEM/SIM/UTILITY ("CHANGE DATABASE Organization") [T];
FILE CARD (TITLE = SCHEMA/ORGANIZATION/NEW);
IF T(VALUE) LSS 0 THEN
  ABORT "Errors detected";
END JOB
```

This example WFL job runs the SIM Utility and directs it to change the schema of the database named ORGANIZATION and to find the new database schema in a file named SCHEMA/ORGANIZATION/NEW. If the SIM Utility detects any errors, the job is aborted.

### Status File

Whenever the SIM Utility is initiated, it creates a status file, which is always named as follows:

DDLUTIL/STATUS/<database name>

The status file is created with the same usercode and pack name as those for the directory file of the database. The status file indicates that the database is currently being accessed by the SIM Utility, and it contains status information about the current SIM Utility run.

When the SIM Utility performs a normal end of task (EOT), it removes the status file regardless of whether any errors were detected. A normal EOT is one in which the SIM Utility terminates after determining that the database is in a safe, consistent mode. After a normal EOT, the database can be processed by a new SIM Utility execution.

The status file prevents conflicting SIM Utility executions and provides restart information in case of a processing interruption, as described in the following paragraphs.

### Conflicting Executions

The status file prevents multiple SIM Utility executions from accessing the same database simultaneously. When the utility first runs, it searches for a status file for the database with which it is to work. If a status file is found and is already in use, the utility terminates, which indicates that another SIM Utility execution is already in progress.

### Program Restarts

The status file is used to provide restart information in case of a halt/load or other interruption. If the SIM Utility finds an unused status file for the database when it runs, the utility attempts to restart the previous SIM Utility run. To do this, the command and input files of the current utility run must match the information saved in the status file. If they do not match, the utility terminates, which indicates that a previous SIM Utility run has not completed. If the command and input files match, the utility restarts the previous SIM Utility run, proceeding from the approximate point at which it was interrupted.

If the SIM Utility is initiated through a WFL job, the Master Control Program (MCP) automatically restarts it after a halt/load. In this case, the same command and input files are given to the utility, and a routine restart should occur.

If the SIM Utility is initiated from a terminal (for example, through CANDE) and is interrupted by a halt/load, the execution can be manually restarted by running the utility again with the same command and input files. For this type of restart to be accomplished, the utility must be reexecuted from the same usercode and with the same FAMILY statement as those of the previous utility run.

Note that when the SIM Utility restarts a previous run, the old DDLRESULTS file is removed. Consequently, the new DDLRESULTS file only contains information corresponding to the restart run.



## Utility Commands

The SIM Utility offers a number of commands for creating and maintaining a SIM database. These commands are shown in the following diagram:

### <utility commands>



Each of these commands is described in the following paragraphs.

## ADD Command

Use the ADD command to create a new SIM database.

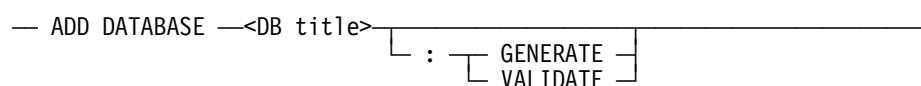
The ADD command requires the presence of an input CARD file that contains an ODL database schema. The schema can include the following kinds of constructs:

- SIM logical constructs (TYPE, CLASS, SUBCLASS, INDEX, VERIFY, ACCESS, or PERMISSION). (Refer to Sections 4 through 8.)
- DMSII physical mapping options (DMSII-OPTIONS). (Refer to Section 9.)
- SIM Utility options for process and output control. (Refer to “Utility Options” later in this section.)
- Dictionary options (DICTIONARY-OPTIONS). (Refer to Section 10).

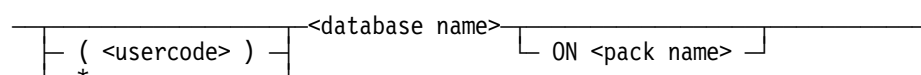
### Syntax

The following diagrams show the syntax for the ADD command:

### <ADD command>



### <DB title>



### Explanation

#### <DB title> Parameter for ADD Command

The <DB title> parameter specifies the name and optionally the location of the new database. The <database name> parameter must be a valid SIM identifier, which can be any combination of letters, digits, hyphens (-), or underscores (\_), starting with a letter and ending with a letter or digit, up to 30 characters in length.

### Caution

Although a SIM database name can be up to 30 characters in length, the physical data management system, DMSII, uses only the first 17 characters. Thus, to avoid accidental destruction of an existing database, make certain that databases under the same usercode do not have database names in which the corresponding first 17 characters are the same.

If either a <usercode> parameter, which can be an asterisk (\*), or a <pack name> parameter or both are specified, they are interpreted identically to any CONTROL FILE ATTRIBUTES usercode or pack name specification in the schema. In fact, if either parameter is specified in both the ADD command and the schema, the two specifications must be identical. This information indicates the location where the directory of the database is to reside, whether the database is being validated or generated. If the database is generated, this information also indicates where the control file is to reside. (Refer to “Control Files” in Section 9, “DMSII Mapping Options,” for more information on CONTROL FILE ATTRIBUTES specifications.)

#### GENERATE Option for ADD Command

If neither the GENERATE option nor the VALIDATE option is specified in the ADD command, the GENERATE option is assumed by SIM. An ADD command with the GENERATE option directs the SIM Utility to create a complete, generated database. For a description of a generated database, refer to “Generated Database” earlier in this section. The utility creates a generated database by performing the following steps:

1. The schema is parsed for correct syntax and semantics.
2. A DMSII DASDL is generated and compiled, causing a DMSII description file to be created.
3. A validated SIM directory file is created.
4. A DMSUPPORT library is compiled.
5. A control file is generated.
6. Database structure files are initialized.
7. If any verifies have been declared, a verify validation phase is performed. This phase ensures that all verifies are valid on the initial, empty database.
8. The directory is stored in the database, and the validated directory file is removed.

If no errors are found, these steps result in a complete, generated database that can be used for all normal functions. Note that the ADD/GENERATE run causes any already existing files with the same names as those for the newly created database to be replaced.

#### VALIDATE Option for ADD Command

An ADD command with the VALIDATE option directs the SIM Utility to create a validated database. For a description of a validated database, refer to “Validated Databases” earlier in this section.

The SIM Utility creates a validated database by performing only steps 1 through 3 indicated under the preceding heading, “GENERATE Option for ADD Command.” Note that, since an ADD/VALIDATE run does not perform step 7 (the verify validation phase), it is possible that a schema that is processed successfully by an ADD/VALIDATE run might fail during an ADD/GENERATE run. Also note that an ADD/VALIDATE run produces a validated directory file as well as a new DMSII description file. These files replace any already existing files with the same names.

An ADD/VALIDATE run can be used to quickly determine the correctness of a database schema for testing purposes. It can be a useful prelude to performing a full ADD/GENERATE run.

## CHANGE Command

Use the CHANGE command to change the schema of an existing generated SIM database.

The CHANGE command requires the presence of an input CARD file that contains an ODL database schema. The schema can contain the same kinds of constructs that are allowed for the ADD command.

#### Syntax

The following diagrams show the syntax for the CHANGE command:

##### <CHANGE command>

```
— CHANGE DATABASE —<DB title> —————|
                                     |
                                     | : |
                                     | | GENERATE
                                     | | VALIDATE
                                     | |
                                     | |
```

##### <DB title>

```
— ( <usercode> ) —————|
| * |
| <database name> |
| ON <pack name> |
|
```

### Explanation

#### <DB title> Parameter for CHANGE Command

The <DB title> parameter for the CHANGE command is interpreted differently than for the ADD command. For the CHANGE command, the <DB title> specifies the current location of the database. If you do not specify a usercode or pack name, SIM performs a standard file search to find the current control file and directory of the database. If the location of the control file and directory differs from CONTROL FILE ATTRIBUTES specifications in the schema, the difference is interpreted as a request to move the control file and directory to the new location specified in the schema. (Refer to “Control Files” in Section 9, “DMSII Mapping Options,” for more information on database schema change with respect to the control file).

#### GENERATE Option for CHANGE Command

If neither the GENERATE option nor the VALIDATE option is specified in the CHANGE command, the GENERATE option is assumed by SIM. A CHANGE command with the GENERATE option directs the SIM Utility to implement the new schema provided, including all necessary logical and physical changes. The result is a complete, generated database. During the CHANGE/GENERATE run, the SIM Utility performs the following steps:

1. The schema is parsed for correct syntax and semantics.
2. The new schema is compared to the old schema to determine the logical schema changes that have been requested. Logical changes are verified for validity and summarized in the DDLRESULTS file.
3. An updated DMSII DASDL file is generated and compiled, resulting in an updated DMSII description file.
4. The new physical database description is examined, and all physical changes are summarized in the DDLRESULTS file.
5. A new, validated SIM directory file is created.
6. If a new DMSUPPORT library is needed, it is compiled.
7. If a database reorganization is needed, SYSTEM/BUILDREORG is called and a reorganization program is compiled.
8. Database applications currently using the database, if any, are temporarily suspended from processing.
9. If any new attribute integrity constraints have been added to the schema (other than new or modified verifies), an attribute validation phase is performed. This phase ensures that all existing data meets the new integrity constraints.
10. If needed, a data cleanup phase is performed. This phase deletes certain data that correspond to deleted constructs.
11. If any file pack names have changed, the affected files are moved to their new locations.
12. If needed, the control file is updated.
13. If needed, new DMSII structures are initialized.

14. If a reorganization is required, the previously compiled reorganization program is initiated.
15. If new DMSII sets or subsets are being generated by the reorganization program, they are tested to ensure that they have been generated correctly.
16. If the schema contains any new or modified verifies, a verify validation phase is performed. This phase ensures that all new or modified verifies are valid against the new database.
17. The new directory is stored in the database, and the validated directory file is removed.
18. Suspended database application programs, if any, are released.
19. If a reorganization program is still in progress, the SIM Utility waits for it to complete.

If all steps are successfully completed, the result of the CHANGE/GENERATE run is a complete, generated database. Whenever possible, the SIM Utility retains the internal timestamps of schema constructs to prevent the need for outstanding queries to be recompiled.

Some important considerations and recommendations relative to performing a CHANGE/GENERATE run are noted in the following list:

- It is strongly recommended that, prior to performing a CHANGE/GENERATE run, the database be completely backed up, including its structure files, description file, audit trail, and DMSUPPORT library. This backup ensures that the database can be reloaded in case the CHANGE command fails and the SIM Utility is unable to recover the database to a consistent state.
- It is recommended that, immediately following a successful CHANGE/GENERATE run, a new backup of the database be made, including its structure files, description file, audit trail, and DMSUPPORT library. This backup ensures that the database can be recovered from any point forward from the time the schema change was made.
- In cases where the SIM Utility detects errors, it attempts to undo any damage that has been done and restore the database to its original state. In some cases, the SIM Utility is not able to do this (for example, when an internal software error occurs). In these cases, the utility issues a message in the DDLRESULTS file, indicating that the database must be recovered manually.
- The database is unavailable to applications during the suspension time that occurs between steps 8 and 18. The suspension time can be a few seconds, several minutes, or, in some cases, several hours. The length of the suspension time depends on the number of tasks that the SIM Utility must perform and the amount of data that must be processed. In particular, steps 9, 10, 11, 15, and 16 have a significant impact on the suspension time. The need for these steps can be determined and the amount of time that each requires can be approximated by performing a CHANGE/VALIDATE run using the new schema and examining the resulting DDLRESULTS file. You can then assess the impact of the new schema and schedule an appropriate time for performing the full CHANGE/GENERATE run.

- After step 18, all existing and new application programs become aware of the database with its new schema. Consequently, some existing queries might be automatically reparsed. In some cases, the new schema can cause existing queries to become invalid. An invalid query receives an error message when it is invoked.
- If a schema change causes a database reorganization, the reorganization program might still be running when application programs are released in step 18, because of the online reorganization capability of DMSII.

In theory, applications can begin to update the database immediately in conjunction with the reorganization. In practice, however, it is recommended that applications not be allowed to update the database until the SIM Utility has successfully completed its run and the new database has been backed up. This practice ensures that the database can be recovered from a backup in case the reorganization fails and the SIM Utility cannot recover the database.

- There is one type of reorganization failure that is not considered an error. If a new DMSII set or subset fails to generate because of a duplicates error, the SIM Utility detects the failure in step 15. As a result, the corresponding SIM construct is simply deleted from the schema and a warning message is generated in the DDLRESULTS file. For example, if an index using the UNIQUE option is added to an existing class, and if the corresponding key attributes contain duplicate data values, the index is deleted from the schema and a warning message is generated. Also, if the UNIQUE option is added to an existing DVA that contains duplicate values, the UNIQUE option is deleted from the schema and a warning message is generated.

### VALIDATE Option for CHANGE Command

In a CHANGE/VALIDATE run, the VALIDATE option causes the SIM Utility to perform only steps 1 through 5 described under the preceding heading, “GENERATE Option for CHANGE Command.” A validated directory file reflecting the new schema is generated. However, the new description file created in step 3 is deleted, and the original description file is left undisturbed.

Some checks that are performed during a CHANGE/GENERATE run are not performed for a CHANGE/VALIDATE run, notably step 16. Therefore, some schemas that pass a CHANGE/VALIDATE run can still fail during a CHANGE/GENERATE run.

The DDLRESULTS file produced by a CHANGE/VALIDATE run describes all logical and physical changes caused by the new schema. Therefore, a CHANGE/VALIDATE run is very useful for determining the effects of a schema change prior to actually implementing the changes through a CHANGE/GENERATE run.



### Explanation

#### <DB title> Parameter for UPDATE Command

The <DB title> parameter specifies the name and optionally the location of the database that is to be updated.

The GENERATE and VALIDATE options are described in the following paragraphs. Using the UPDATE command only to update a database to current software levels is described under the heading “Updating Software Levels,” later in this section. Using the UPDATE command in a procedure that involves copying a database to a new location and updating the copied database is described subsequently under the heading “Copying a Database.”

#### GENERATE Option for UPDATE Command

If you do not specify GENERATE or VALIDATE, the GENERATE option is assumed by SIM. The GENERATE option causes the SIM Utility to fully update the database to the current levels of DMSII and SIM software. An UPDATE/GENERATE run causes the SIM Utility to perform the following steps:

1. The CARD file, if supplied, is parsed for valid syntax and semantics.
2. The current directory of the database is loaded. If the database currently uses an old directory level compared to the current SIM software, either an error message is generated (if the level is too old) or the directory is migrated to the current level.
3. The current schema of the database is extracted and reparsed.
4. A new DASDL is generated and compiled, resulting in a new DMSII description file.
5. A new, validated SIM directory file is created.
6. A new DMSUPPORT library is compiled.
7. A new control file is generated.
8. The validated directory is stored in the database, and the validated directory file is removed.

If any errors are detected, the UPDATE command is aborted and the database is recovered to its original state. However, as with the CHANGE command, it is recommended that the database be completely backed up prior to performing the UPDATE/GENERATE run.



Several important differences between the UPDATE command and the CHANGE command are summarized as follows:

- The CHANGE command requires that the database to which it is directed be compatible with the current level of SIM software. If the SIM Utility detects that the database is not compatible with the software, it does not process the CHANGE command and it issues an error message indicating that an UPDATE command is required.
- The UPDATE command unconditionally compiles a new DMSUPPORT library and generates a new control file, thereby ensuring that the database is updated to the current level of DMSII software. In contrast, the CHANGE command compiles a new DMSUPPORT library and generates a new control file only as needed.
- The UPDATE command only accepts DMSII global options as input schema, whereas the CHANGE command accepts a complete ODL schema. Therefore, the UPDATE command does not allow changes to the logical schema or to individual mapping options.
- The UPDATE command assumes that the database might have been copied. It therefore does not rely on existing pack names stored in the SIM directory and the DMSII description file. Therefore, the UPDATE command does not take any action to copy or move any files. For this reason, when a database is copied, it must be placed on the desired packs prior to the initiation of the UPDATE command. In contrast, the CHANGE command compares old and new pack names and interprets any differences as a request to move the corresponding files.

#### VALIDATE Option for UPDATE Command

The VALIDATE option causes the SIM Utility to perform only steps 1 through 5 described under the preceding heading, "GENERATE Option for UPDATE Command." As a result, a new validated directory is generated. A new DMSII description file is created per step 4, but it is not saved.

You can use an UPDATE/VALIDATE run to verify the input CARD file and to ensure that the DASDL of the database can be recompiled prior to performing an UPDATE/GENERATE run.

## Updating Software Levels

When the UPDATE command is used only to update the database to the current levels of SIM and DMSII software, the CARD file should be omitted, or it should contain the same DMSII global options that were used in the latest ADD/GENERATE or CHANGE/GENERATE run, whichever was more recent.

## Copying a Database

The UPDATE command can be used in the following procedure for copying and updating a database:

1. Copy the database, including its structure files, audit trail, control file, and description file, to its new location. The DMSUPPORT library need not be copied since it is to be regenerated.
2. Create a schema file that is to be included in the CARD file for the UPDATE/GENERATE run. This schema file should contain DMSII global options that reflect the characteristics of the copied database, such as DMSUPPORT title, OPTIONS, PARAMETERS, and pack names.
3. Run the SIM Utility using the UPDATE command with the GENERATE option, and file-equate the CARD file to use the schema file that you created in step 2.

When the SIM Utility completes, the database can be accessed by application programs. If a CHANGE command is subsequently performed for the database, the database schema should contain the same DMSII global options that were prepared in step 2, except where changes to these options are actually desired.

## REINITIALIZE Command

Use the REINITIALIZE command to reinitialize all user data for a given database, that is, to cause all entities to be deleted from user classes and all class attributes to be set to null.

The REINITIALIZE command normally does not use a CARD file. However, use of a CARD file with this command is optional to accommodate the possible need for certain SIM Utility options.

### Syntax

The following diagrams show the syntax for the REINITIALIZE command:

#### <REINITIALIZE command>

— REINITIALIZE DATABASE —<DB title>—————|

#### <DB title>

—|—————<database name>—————|  
| ( <usercode> ) |  
| \* | ON <pack name> |

### Explanation

#### <DB title> Parameter for REINITIALIZE Command

The <DB title> parameter specifies the name and optionally the location of the database that is to be reinitialized.

## Using the REINITIALIZE Command

The REINITIALIZE command is similar to the *INITIALIZE* = function offered by the DMUTILITY program for DMSII databases. However, rather than simply deleting all existing data, the REINITIALIZE command performs additional checks and updates to achieve a valid, consistent SIM database. The need for these additional checks and updates is the reason that a DMUTILITY *INITIALIZE* statement is not allowed for generated SIM databases.

Note that you cannot use the REINITIALIZE command if the database contains any verifies. An explanation of this restriction and a description of methods that you can use to reinitialize a database that contains verifies are presented in the following paragraphs.

## Reinitializing a Database that Contains Verifies

The SIM Utility does not allow the REINITIALIZE command if the database contains any verifies, for the following reason: the resultant empty database might not be valid against existing verifies, and the SIM Utility cannot test this condition in a manner that allows it to abort the reinitialization if such action is needed.

Therefore, to reinitialize a database that contains verifies, you must use either of the following two methods:

- Remove all verifies from the database by performing a CHANGE/GENERATE run, perform the REINITIALIZE run, and then add the verifies back to the database with another CHANGE/GENERATE run.
- Manually reinitialize all data by performing the appropriate OML update queries. Set class attributes to null by performing following query for each class attribute in the database:

```
MODIFY <class name> (<class attribute name>:= EXCLUDE)
```

Delete all entities by performing the following query for each base class in the database:

```
DELETE LIMIT=ALL <class name> WHERE TRUE
```

For information on the use of OML, refer to the *SIM OML Programming Guide*.

## LIST Command

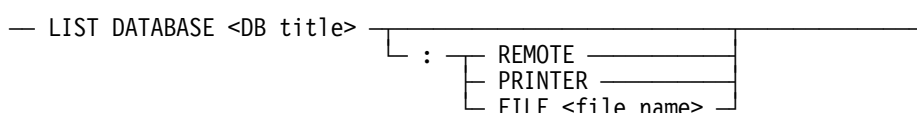
Use the LIST command to list the schema of an existing database. This command generates a syntactically correct database schema that includes DMSII mapping options.

The LIST command normally does not use a CARD file. However, use of a CARD file with this command is optional to allow SIM Utility options to be invoked.

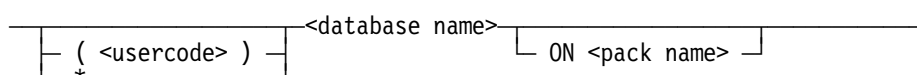
### Syntax

The following diagrams show the syntax for the LIST command:

#### <LIST command>



#### <DB title>



### Explanation

The <DB title> parameter specifies the name and optionally the location of the database for which the schema is to be listed.

If you specify the REMOTE option, the LIST command sends output to a remote file. REMOTE is the default if the SIM Utility is run from a terminal.

If you specify the PRINTER option, the LIST command produces a printer listing. PRINTER is the default if the SIM Utility is run in batch mode.

If you specify the FILE option, the file designated in the <file name> parameter is created as a disk file.

Note that when a LIST command is performed on a viewed database (refer to “Viewed Database” earlier in this section), certain DMSII mapping options might appear that are not actually allowed for SIM databases generated by ODL. Therefore, if a LIST schema derived from a viewed database is submitted to the SIM Utility through an ADD command or a CHANGE command, processing errors might result.

## LOAD Command

Use the LOAD command to load the schema of a database into an ADDS dictionary.

The CARD file must include a dictionary physical option specification. You can request, by using the EXCLUSIVE = TRUE option, that the database be loaded in exclusive transaction state. For more information on loading databases into an ADDS dictionary, refer to the *InfoExec ADDS Operations Guide*.

The following diagrams show the syntax for the LOAD command:

### <LOAD command>

```
— LOAD DATABASE —<DB title> — : — EXCLUSIVE — = — TRUE —
                                     FALSE —
```

### <DB title>

```
— ( <usercode> ) — <database name> — ON <pack name> —
  * —
```

### Explanation

The <DB title> parameter specifies the name and optionally the location of the database for which the schema is to be loaded.

If you specify the EXCLUSIVE = TRUE option, the database is loaded into the ADDS dictionary in exclusive transaction state.

# Utility Files

The SIM Utility uses a number of files to obtain input beyond the utility command parameter and to produce output. Some of these files can be file-equated to direct the SIM Utility to find or create specific files. The utility files that can be file-equated are the following:

- CARD file (primary input file)
- DASDL file (file used to create and access description files)
- DDLRESULTS file (summary of the SIM Utility execution)
- LINE file (file used to create printer listings)

These utility files are described in the following paragraphs.

## CARD File

The CARD file is the primary input file for the SIM Utility. It is required for the ADD and CHANGE commands, and it is optional for the UPDATE, REINITIALIZE, and LIST commands.

By default, the SIM Utility expects the CARD file to be a disk file named CARD, visible within the FAMILY specification with which the SIM Utility is initiated. It can be file-equated to use a specific title, including a pack name, or it can be file-equated to use a file kind other than disk. For example, it can be file-equated to use a remote file if the input is relatively small and can be entered from a terminal.

Only the first 72 characters of each CARD file record are scanned by the SIM Utility. The remaining eight characters can contain sequence numbers, or they can be blank. It is advantageous to use a CARD file that contains sequence numbers. The SIM Utility can then relate error and warning messages back to the sequence numbers corresponding to the syntax that caused the messages to occur. For this reason, the CARD file given to the SIM Utility should be a disk file with a FILEKIND of SEQDATA, TEXTDATA, or some other type that uses a 72-character text, 8-character sequence number format.

## DASDL File

The DASDL file is both an input file and an output file. For the ADD, CHANGE, and UPDATE commands, the DASDL file is used to create a new description file. For the CHANGE and UPDATE commands, it is also used to access an existing description file. The DASDL file is not used for the REINITIALIZE or LIST commands.

In the process of accessing an existing description file, the SIM Utility normally uses a standard file search to seek a file named *DESCRIPTION/<database name>*. That is, the file is sought under the usercode for the utility, and it is also sought as a nonusercoded (\*) file. Moreover, the file is sought under the primary and secondary family names for the utility. When a description file is found, the utility verifies that the file matches the database with which the utility is working.

To ensure that the SIM Utility finds the correct description file for a CHANGE command or an UPDATE command, it is suggested that the DASDL file be file-equated to the correct title, including the correct usercode and pack name. The following example shows a CANDE *RUN* command that file-equates a DASDL file in this manner:

```
RUN *SYSTEM/SIM/UTILITY ("CHANGE DATABASE Organization");
FILE CARD (TITLE = SCHEMA/ORGANIZATION/NEW);
FILE DASDL (TITLE = *DESCRIPTION/ORGANIZATION ON PACK);
```

When the SIM Utility creates a new description file it uses the same DASDL title with which it found the input description file. A successful CHANGE or UPDATE run replaces the old description file with a new one.

For the ADD command, the DASDL file is used only to create a new description file; there is no input description file in this case. If the DASDL file is file-equated, the specified name is used. Otherwise, the new description file is created as *DESCRIPTION/<database name>* under the usercode and primary family for the SIM Utility.

## DDLRESULTS File

Every time the SIM Utility runs, it creates a DDLRESULTS file that summarizes the execution of the utility. By default, this file is named *DDLRESULTS/<database name>*. It resides under the usercode and primary family with which the utility is initiated. If desired, the DDLRESULTS file can be file-equated so that it is created with a different name.

Note that, on rare occasions, the SIM Utility might terminate before it has a chance to lock the DDLRESULTS file with the correct title. When this action occurs, the file simply is given the name DDLRESULTS rather than the aforementioned default name or a file-equated name.

The format of the DDLRESULTS file is compatible with the compiler error file format. That is, error and warning messages are prefixed in a similar manner as those produced by compilers. One advantage of this feature is that the DDLRESULTS file can be loaded by the Editor utility (SYSTEM/EDITOR) as an error file and later be used to examine error and warning messages. For example, you can use this feature by performing the following steps:

1. Through CANDE, enter the following (as a single command) to perform a SIM Utility run:

```
RUN *SYSTEM/SIM/UTILITY ("ADD DATABASE TESTDB");
FILE SCHEMA=SCHEMA/TESTDB
```

The SIM Utility detects warnings and errors as indicated in its final display message, and it creates a file named DDLRESULTS/TESTDB.

2. Enter the following to load the schema file as a work file and initiate SYSTEM/EDITOR:

```
GET SCHEMA/TESTDB;U *SYSTEM/EDITOR
```

3. After the Editor utility comes up, load the DDLRESULTS file as an error file by entering the following:  

```
]LOAD E DDLRESULTS/TESTDB
```
4. Use the ]+ERROR and ]-ERROR commands to view the error and warning messages in context. By using these commands, you can display each of these messages and concurrently move the cursor to the line of the schema work file relevant to the message.

This feature makes it very easy to examine and correct errors in the schema file. Note that you can only view error and warning messages through the error file mechanism of the Editor. Other informational messages are not displayed. Note also that some messages might be too long to be viewed on the top line of the display. You can use the Editor WINDOW command to view full message text. Refer to the *Editor Operations Guide* for more information on the error file mechanism of the Editor.

## LINE File

The LINE file is used by the SIM Utility to create printer listings. A printer listing is created in either of the following two situations:

- When the CARD file contains a LIST option, all CARD images (starting with the record that contains the LIST option) are printed to the LINE file until the CARD file is exhausted or the LIST option is reset.
- When a LIST command is given with the PRINTER option (which is the default when the SIM Utility is initiated in batch mode), the database schema is printed to the LINE file.

If the LIST option is used in conjunction with a LIST command given with the PRINTER option, the two printouts are directed to a single LINE file. The LINE file is created without any carriage controls; therefore, it can be file-equated to a disk file, if desired. It can also be equated to use a specific TRAINID or other printer file attributes.

For more information about the LIST option, refer to “LIST Option” later in this section. For more information about the LIST command, refer to “LIST Command” earlier in this section.



## Utility Options

The SIM Utility provides the following options for process and output control:

- LIMIT
- LIST
- WARNSUPR

These utility options are described in the following paragraphs. The options can be specified in the CARD file whenever it is used. Except as noted, each option can be set or reset anywhere in the CARD file, and each option can be specified any number of times.

### LIMIT Option

The default error limit for the SIM Utility is 150. You can use the LIMIT option to override this default limit. When the error limit is exceeded, the SIM Utility terminates processing. Before the current error limit is exceeded, you can specify the LIMIT option any number of times. The LIMIT option need not be preceded by SET, and it cannot be reset.

To set the LIMIT option, use the following syntax:

```
— $ [ SET ] LIMIT = <number> _____|
```

### LIST Option

When the LIST option is set, the SIM Utility lists the given CARD file to a printer listing using the LINE file (refer to “LINE File” earlier in this section). The listing contains, in addition to the CARD file, any error, warning, and notice messages. Note that the SIM Utility never sets the LIST option by default, even when the utility is executed in batch mode.

To set the LIST option, use the following syntax:

```
— $ SET LIST _____|
```

To reset the LIST option, use the following syntax:

```
— $ RESET LIST _____|
```

### WARNSUPR Option

When the WARNSUPR option is set, warning messages (such as those indicating reserved-word conflicts) are suppressed in the DDLRESULTS and LINE files.

To set the WARNSUPR option, use the following syntax:

```
— $ SET WARNSUPR _____|
```

To reset the WARNSUPR option, use the following syntax:

```
— $ RESET WARNSUPR _____|
```

# Section 12

## Changing a SIM Database Schema

This section identifies the kinds of schema changes to a SIM database that are allowed by SIM. Special restrictions or implications are noted, where applicable, for allowed changes. The section also identifies schema changes that are not allowed by SIM.

The information is presented in tables and notes under the following headings:

- User-Defined Type Changes
- Base Class Changes
- Subclass Changes
- Class Attribute Changes
- Data-Valued Attribute (DVA) Changes
- Entity-Valued Attribute (EVA) Changes
- Index Changes
- Verify Changes
- Access Changes
- Permission Changes
- DMSII Mapping Option Changes
- Dictionary Option Changes

Each table includes three columns. The first column lists the kinds of changes. The second column indicates, for each kind of change, whether it is allowed (Yes) or not allowed (No) by SIM. The third column identifies, by number, the notes following the table that relate to the corresponding allowed changes.

This section concludes with information on how you might circumvent some schema change restrictions.

# Understanding Terms

The following definitions are provided to help you understand the meaning of specific terms used in this section. (Note that the examples presented to illustrate immediate type, base type, and type chain do not reflect the ORGANIZATION database.)

| Term           | Definition  |
|----------------|---|
| Data type      | The type or range of a user-defined type, a compound component, a class attribute, or a data-valued attribute (DVA). Possible data types are primitive, constructor, and user-defined. Additionally, the data type for a DVA can be SUBROLE.  |
| Immediate type | <p>The first data type referenced in the definition of a user-defined type, a compound component, a class attribute, or a DVA. For example, consider the following declarations:</p> <pre>TYPE Color    = SYMBOLIC (red, blue, green, pink); TYPE Primary  = Color (red, blue, green); CLASS Car     (body-color: Color;      interior-color: Primary;     );</pre> <p>In this example, the immediate type of Color is SYMBOLIC, the immediate type of both Primary and body-color is Color, and the immediate type of interior-color is Primary.</p> |
| Base type      | The most basic data type of a user-defined type, a compound component, a class attribute, or a DVA. In the previous example, the base type of Color, Primary, body-color, and interior-color is SYMBOLIC.   |
| Type chain     | The set of data types from an immediate type to a base type. In the previous example, the type chain of interior-color includes Primary, the subrange (red, blue, green), Color, and SYMBOLIC.  |

## How Schema Changes Affect Queries

When you change the schema of a database, existing queries that are used by application programs and ad hoc query programs might be affected. In some cases, existing queries might need to be modified in order to be valid with respect to the new schema. In other cases, queries might be automatically reparsed the next time they are invoked. Of course, some queries might not be affected at all by a schema change.

Whenever a query is parsed, SIM retains sufficient information about the query to determine if and when it is invalid or requires reparsing. SIM assesses the validity of a query each time that query is executed. If a query is considered invalid, it receives an error. If a query requires reparsing, SIM automatically reparses the query and returns a warning indicating that reparsing occurred. For host language interface (HLI) programs, the reparse warning indicates that the program could achieve slightly better performance if it were recompiled, thereby eliminating the need for automatic reparsing.

The following is a summary of the conditions under which queries might become affected by a schema change:

- When a construct is deleted, all queries that reference the deleted construct must be modified and resubmitted.
- When a class is changed, all queries that reference the class are automatically reparsed the next time they are invoked. A class is considered changed when any of its attributes, indexes, verifies, or accesses are added, deleted, or modified, or when any of its subclasses are added or deleted.
- A subclass might be considered modified if a change is made to one of its superclasses. For example, consider a class and a subclass that are mapped to the same variable-format data set. If an attribute is added to or deleted from the class, the subclass also is marked as changed, owing to the physical mapping properties of variable-format data sets.
- Changes to physical mapping options made by means of DMSII mapping-options declarations do not affect existing queries. Also, changes made by means of permission declarations do not affect existing queries.

# User-Defined Type Changes

The following considerations apply to changing user-defined types in a SIM database.

| User-Defined Type Change                         | Allowed? | Notes |
|--|----------|-------|
| Add new TYPE.                                    | Yes      |       |
| Delete TYPE.                                     | Yes      | 1     |
| Change TYPE name.                                | No       |       |
| Change TYPE base type.                           | Yes      | 2     |
| Change TYPE subrange.                            | Yes      | 2     |
| Add TYPE subrange.                               | Yes      | 2     |
| Remove TYPE subrange.                            | Yes      | 2     |
| <b>If immediate type is a user-defined TYPE:</b> |          |       |
| Change to a different user-defined TYPE.         | Yes      | 2     |
| Remove use of user-defined TYPE.                 | Yes      | 2     |
| Add a new user-defined TYPE into type chain.     | Yes      | 2     |
| <b>If base type is NUMBER:</b>                   |          |       |
| Change NUMBER format/size.                       | Yes      | 2     |
| <b>If base type is STRING:</b>                   |          |       |
| Change STRING size.                              | Yes      | 2     |
| Change STRING from FIXED to VARIABLE.            | Yes      | 2     |
| Change STRING from VARIABLE to FIXED.            | Yes      | 2     |
| <b>If base type is Compound:</b>                 |          |       |
| Add component.                                   | Yes      |       |
| Change component name.                           | No       |       |
| Change component type.                           | Yes      | 2     |
| Delete component.                                | Yes      | 2     |

continued

| User-Defined Type Change                       | Allowed? | Notes |
|--|----------|-------|
| <b>If base type is SYMBOLIC:</b>               |          |       |
| Add value to SYMBOLIC.                         | Yes      | 2     |
| Change name of existing SYMBOLIC value.        | No       |       |
| Change unordered SYMBOLIC to ORDERED SYMBOLIC. | Yes      | 2     |
| Change ORDERED SYMBOLIC to unordered SYMBOLIC. | Yes      | 2     |
| Delete value from SYMBOLIC.                    | Yes      | 2     |

**Notes:**

1. All related *TYPEs*, *DVAs*, compound components, and class attributes must also be deleted or changed so that they do not reference the deleted *TYPE*.
2. If the *TYPE* is used by any existing *DVA*, compound component, or class attribute, the *TYPE* change is considered to be a change to that *DVA*, compound component, or class attribute, and any applicable change restrictions for the *DVA*, compound component, or class attribute applies.

## Base Class Changes

The following considerations apply to changing base classes in a SIM database.

| Base Class Change       | Allowed? | Notes |
|-------------------------|----------|-------|
| Add new base class.     | Yes      |       |
| Delete base class.      | Yes      | 1     |
| Change base class name. | No       |       |

**Note:**

1. All related subclasses, indexes, verifies, and accesses and all referencing *EVA*s must also be deleted. The attributes and class attributes of the base class are automatically deleted.

### Subclass Changes

The following considerations apply to changing subclasses in a SIM data base.

| Subclass Change                           | Allowed? | Notes |
|---|----------|-------|
| Add new subclass.                         | Yes      |       |
| Delete subclass.                          | Yes      | 1     |
| Change subclass name.                     | No       |       |
| Add new superclass to existing subclass.  | No       |       |
| Delete superclass from existing subclass. | No       |       |

**Note:**

1. *All subordinate subclasses and related indexes, verifies, and accesses and all referencing EVAs must also be deleted. The attributes and class attributes of the subclass are automatically deleted.*

### Class Attribute Changes

The following considerations apply to changing class attributes in a SIM database.

| Class Attribute Change                           | Allowed? | Notes |
|--|----------|-------|
| Add new class attribute.                         | Yes      |       |
| Delete class attribute.                          | Yes      |       |
| Change class attribute name.                     | No       |       |
| Change class attribute base type.                | No       |       |
| Add subrange.                                    | Yes      | 1     |
| Change subrange.                                 | Yes      | 1     |
| Delete subrange.                                 | Yes      |       |
| <b>If immediate type is a user-defined TYPE:</b> |          |       |
| Change to a different user-defined TYPE.         | Yes      |       |
| Remove use of user-defined TYPE.                 | Yes      |       |
| Add a new user-defined TYPE into type chain.     | Yes      |       |



continued

| Class Attribute Change                         | Allowed? | Notes |
|--|----------|-------|
| <b>If base type is NUMBER:</b>                 |          |       |
| Change NUMBER format/size.                     | Yes      |       |
| <b>If base type is STRING:</b>                 |          |       |
| Change STRING size.                            | Yes      |       |
| Change STRING from FIXED to VARIABLE.          | No       |       |
| Change STRING from VARIABLE to FIXED.          | Yes      |       |
| <b>If base type is Compound:</b>               |          |       |
| Add component.                                 | Yes      |       |
| Change component name.                         | No       |       |
| Change component type.                         |          | 2     |
| Delete component.                              | Yes      |       |
| <b>If base type is SYMBOLIC:</b>               |          |       |
| Add value to SYMBOLIC.                         | Yes      | 3     |
| Change name of existing SYMBOLIC value.        | No       |       |
| Change unordered SYMBOLIC to ORDERED SYMBOLIC. | Yes      |       |
| Change ORDERED SYMBOLIC to unordered SYMBOLIC. | Yes      |       |
| Delete value from SYMBOLIC.                    | No       |       |

**Notes:**

1. *If a subrange is added or changed, the class attribute must satisfy the new subrange restriction.*
2. *The rules that apply to a class attribute also apply to the component of a compound class attribute.*
3. *New symbolic values can be added only to the end of existing symbolic values.*

### Data-Valued Attribute (DVA) Changes

The following considerations apply to changing data-valued attributes (DVAs) in a SIM database.

| DVA Change                               | Allowed? | Notes |
|--|----------|-------|
| Add new DVA.                             | Yes      | 1     |
| Delete DVA.                              | Yes      |       |
| Change DVA name.                         | No       |       |
| Change DVA base type.                    | No       |       |
| <b>Change single-valued DVA options:</b> |          |       |
| Add UNIQUE option.                       | Yes      | 2, 12 |
| Remove UNIQUE option.                    | Yes      |       |
| Add REQUIRED option.                     | Yes      | 3     |
| Remove REQUIRED option.                  | Yes      |       |
| Add READONLY option.                     | Yes      |       |
| Remove READONLY option.                  | Yes      |       |
| Change cardinality to multivalued.       | No       |       |
| Add INITIALVALUE option.                 | Yes      |       |
| Change INITIALVALUE option.              | Yes      |       |
| Remove INITIALVALUE option.              | Yes      |       |
| <b>Change multivalued DVA options:</b>   |          |       |
| Add UNIQUE option.                       | Yes      | 2, 4  |
| Remove UNIQUE option.                    | Yes      |       |
| Add REQUIRED option.                     | Yes      | 3     |
| Remove REQUIRED option.                  | Yes      |       |
| Add READONLY option.                     | Yes      |       |
| Remove READONLY option.                  | Yes      |       |
| Change cardinality to single valued.     | No       |       |
| Add DISTINCT option.                     | Yes      | 4, 5  |
| Remove DISTINCT option.                  | Yes      |       |
| Add MAX <limit> option.                  | Yes      | 6     |

continued

| DVA Change                                       | Allowed? | Notes |
|--|----------|-------|
| Change MAX <limit> option.                       | Yes      | 6     |
| Remove MAX <limit> option.                       | Yes      | 7     |
| Add INITIALVALUE option.                         | No       |       |
| Add subrange.                                    | Yes      | 8     |
| Delete subrange.                                 | Yes      |       |
| Change subrange.                                 | Yes      | 8     |
| <b>If immediate type is a user-defined TYPE:</b> |          |       |
| Change to a different user-defined TYPE.         | Yes      |       |
| Remove use of user-defined TYPE.                 | Yes      |       |
| Add a new user-defined TYPE into type chain.     | Yes      |       |
| <b>If base type is NUMBER:</b>                   |          |       |
| Change NUMBER format/size.                       | Yes      |       |
| <b>If base type is STRING:</b>                   |          |       |
| Change STRING size.                              | Yes      |       |
| Change STRING from FIXED to VARIABLE.            | No       |       |
| Change STRING from VARIABLE to FIXED.            | Yes      |       |
| <b>If base type is Compound:</b>                 |          |       |
| Add component.                                   | Yes      | 9     |
| Change component name.                           | No       |       |
| Change component type.                           |          | 10    |
| Delete component.                                | Yes      | 9     |

continued

| DVA Change                                     | Allowed? | Notes |
|--|----------|-------|
| <b>If base type is SYMBOLIC:</b>               |          |       |
| Add value to SYMBOLIC.                         | Yes      | 11    |
| Change name of existing SYMBOLIC value.        | No       |       |
| Change unordered SYMBOLIC to ORDERED SYMBOLIC. | Yes      |       |
| Change ORDERED SYMBOLIC to unordered SYMBOLIC. | Yes      |       |
| Delete value from SYMBOLIC.                    | No       |       |
| <b>If base type is SUBROLE:</b>                |          |       |
| Add new subclass.                              | Yes      |       |
| Delete existing subclass.                      | Yes      |       |

**Notes:**

1. If added to an existing class, the DVA cannot have an *INITIALVALUE* specified, and it cannot be *REQUIRED*.
2. The data values currently used by the DVA must be unique. If they are not unique, the *UNIQUE* option is removed from the schema.
3. The DVA must not be null for any existing entities.
4. The multivalued DVA must currently be mapped to a disjoint data set.
5. The data values currently used by the DVA must be distinct. If they are not, the *DISTINCT* option is removed from the schema.
6. The multivalued DVA must meet the new *MAX* limit. Also, if the multivalued DVA is mapped to an occurring item, the *MAX* limit can only be increased and the *MAX* limit cannot exceed 4095.
7. The multivalued DVA must not be mapped to an occurring item.
8. The DVA must satisfy the new or modified subrange condition.
9. If the component is an element of a multivalued DVA, the DVA must not be mapped to an embedded data set that has a spanning set.
10. The rules that apply to a DVA also apply to the component of a compound DVA.
11. New symbolic values can be added only to the end of existing symbolic values.
12. If the owner of the DVA is a subclass, either the subclass must be mapped to a disjoint data set, or the DVA must already be the key of an existing index.

## Entity-Valued Attribute (EVA) Changes

The following considerations apply to changing entity-valued attributes (EVAs) in a SIM database.

| EVA Change                               | Allowed? | Notes |
|--|----------|-------|
| Add new EVA.                             | Yes      |       |
| Delete EVA.                              | Yes      | 1     |
| Change EVA name.                         | No       |       |
| Change EVA target.                       | No       |       |
| <b>Change single-valued EVA options:</b> |          |       |
| Add UNIQUE option.                       | No       |       |
| Remove UNIQUE option.                    | No       |       |
| Add REQUIRED option.                     | Yes      | 2     |
| Remove REQUIRED option.                  | Yes      |       |
| Add READONLY option.                     | Yes      |       |
| Remove READONLY option.                  | Yes      |       |
| Change cardinality to multivalued.       | No       |       |
| Change INVERSE.                          | Yes      | 3     |
| <b>Change multivalued EVA options:</b>   |          |       |
| Add UNIQUE option.                       | No       |       |
| Remove UNIQUE option.                    | No       |       |
| Add REQUIRED option.                     | Yes      | 2     |
| Remove REQUIRED option.                  | Yes      |       |
| Add READONLY option.                     | Yes      |       |
| Remove READONLY option.                  | Yes      |       |
| Change cardinality to single-valued.     | No       |       |
| Add DISTINCT option.                     | No       |       |
| Remove DISTINCT option.                  | No       |       |
| Add MAX option.                          | Yes      | 4     |
| Remove MAX option.                       | Yes      |       |
| Change MAX limit.                        | Yes      | 4     |
| Change INVERSE.                          | Yes      | 3     |

**Notes:**

1. All references to the EVA, such as by an *INVERSE* option or by an *ACCESS*, must also be deleted.
2. The EVA must not be null for any existing entities.
3. The only change allowed to the *INVERSE* of an EVA is as follows: If the *INVERSE* on an existing EVA is unspecified, the inverse EVA can be added to the appropriate structure and designated as the *INVERSE*. This action merely names the previously unnamed *INVERSE*. The newly named EVA must retain the cardinality and other attribute options of the previously unnamed EVA.
4. The multivalued EVA must currently satisfy the new or modified *MAX* limit.

## Index Changes

The following considerations apply to changing indexes in a SIM database.

| Index Change  | Allowed? | Notes |
|---|----------|-------|
| Add new index on new class.   | Yes      |       |
| Add new index on existing base class.   | Yes      | 1     |
| Add new index on existing subclass where key is single multivalued DVA.         | Yes      | 1     |
| Add new index on existing subclass where key is one or more single-valued DVAs. | Yes      | 2     |
| Delete index.   | Yes      |       |
| Change index name.  | No       |       |
| Change index target.  | No       |       |
| Change index key.   | Yes      | 3     |
| Add <i>UNIQUE</i> option.   | Yes      | 4     |
| Remove <i>UNIQUE</i> option.  | Yes      |       |

**Notes:**

1. The index key can be a multivalued DVA only if the DVA is already mapped to a disjoint data set. Also, if the index is designated as *UNIQUE*, existing data values used by the index key must be unique; if they are not unique, the new index is not added.
2. If the subclass is mapped to a variable format, the attributes of the index key must be mapped to the fixed part of the record. Therefore, the key attributes must be added along with the index, or they must already be key attributes of another index.
3. The current and new keys of the index must consist of single-valued DVAs, and each key must adhere to the rule in note 2.
4. The data values used by the index key must be unique. If they are not unique, the *UNIQUE* option is ignored.

## Verify Changes

The following considerations apply to changing verifies in a SIM database.

| Verify Change                    | Allowed? | Notes |
|----------------------------------|----------|-------|
| Add new verify.                  | Yes      | 1     |
| Delete verify.                   | Yes      |       |
| Change verify name.              | No       |       |
| Change verify perspective class. | No       |       |
| Change verify condition.         | Yes      | 1     |
| Change verify error message.     | Yes      |       |

**Note:**

1. *A new or modified verify cannot be submitted in the same schema change that requires a database reorganization. Also, the database must meet the new or modified verify condition. If it does not, the schema change is aborted.*

## Access Changes

The following considerations apply to changing accesses in a SIM database.

| Access Change                       | Allowed? |
|-------------------------------------|----------|
| Add new access.                     | Yes      |
| Delete access.                      | Yes      |
| Change access name.                 | No       |
| Change access target.               | No       |
| Change access attributes.           | Yes      |
| Change access OML verbs.            | Yes      |
| Change access selection expression. | Yes      |

### Permission Changes

The following considerations apply to changing permissions in a SIM database.

| Permission Change                     | Allowed? |
|---------------------------------------|----------|
| Add new permission.                   | Yes      |
| Delete permission.                    | Yes      |
| Change permission name.               | No       |
| Change permission access constraints. | Yes      |
| Change permission accesses.           | Yes      |

### DMSII Mapping Option Changes

The following considerations apply to changing DMSII mapping options in a SIM database.

| DMSII Option Change                                      | Allowed? | Notes |
|--|----------|-------|
| Change global option.                                    | Yes      | 1     |
| <b>Change class option for existing base class:</b>      |          |       |
| Change surrogate option.                                 | No       |       |
| Change data set type option.                             | No       |       |
| Change data set physical option.                         | Yes      |       |
| <b>Change subclass option for existing subclass:</b>     |          |       |
| Change subclass mapping type.                            | No       |       |
| Change data set type option.                             | No       |       |
| Change data set physical option.                         | Yes      |       |
| <b>Change single-valued DVA option for existing DVA:</b> |          |       |
| Change size option.                                      | No       |       |
| Change stored option.                                    | No       |       |



continued

| DMSII Option Change                                      | Allowed? | Notes |
|--|----------|-------|
| <b>Change multivalued DVA option for existing DVA:</b>   |          |       |
| Change multivalued DVA mapping option.                   | No       |       |
| Change data set type option.                             | No       |       |
| Change data set physical option.                         | Yes      |       |
| Change set type option.                                  | No       |       |
| Change set-subset physical option.                       | Yes      |       |
| <b>Change single-valued EVA option for existing EVA:</b> |          |       |
| Change single-valued EVA mapping option.                 | No       |       |
| <b>Change multivalued EVA option for existing EVA:</b>   |          |       |
| Change multivalued EVA mapping option.                   | No       |       |
| Change set type option.                                  | No       |       |
| Change set-subset physical option.                       | Yes      |       |
| <b>Change index option for existing index:</b>           |          |       |
| Change duplicate options.                                | No       |       |
| Change set type option.                                  | No       |       |
| Change set-subset physical option.                       | Yes      |       |

**Note:**

1. *Certain database options are set automatically by SIM and cannot be reset by the user.*

### Dictionary Option Changes

The following considerations apply to changing dictionary options.

| Dictionary Option Change             | Allowed? | Notes |
|--------------------------------------|----------|-------|
| Add dictionary or directory name.    | Yes      |       |
| Delete dictionary or directory name. | Yes      | 1     |
| Change dictionary name.              | Yes      |       |
| Change directory name.               | Yes      |       |

**Note:**

1. *After dictionary options are first used, the database remains active with respect to ADDS. The dictionary-options declaration can be removed from the schema, but the database cannot be disconnected from ADDS.*

### Circumventing Schema Change Restrictions

As shown by the preceding tables, SIM does not allow you to make certain types of schema changes directly. However, you can sometimes achieve a desired schema change that cannot be made directly by performing a series of steps. For example, consider the following schema:

```
CLASS Employee
  (hire-date: DATE
  );
```

Suppose you wish to change the single-valued DVA hire-date to a multivalued DVA. Such a change is not directly allowed by SIM. However, you can achieve the change by performing the following steps:

1. Perform a CHANGE DATABASE run that adds a new multivalued DVA named employee-hire-dates to the Employee base class.
2. Perform a query that copies existing hire-date values to employee-hire-dates. For example, the following OML query could be entered through a query product such as the Interactive Query Facility (IQF):

```
MODIFY LIMIT=ALL Employee
  (employee-hire-dates:= INCLUDE (hire-date))
WHERE TRUE
```

3. Perform a second CHANGE DATABASE run that deletes the old DVA, hire-date.

By using a combination of schema changes and OML queries, you can effectively change attribute cardinality and mapping types, and you can perform other types of schema changes that are disallowed in a single schema change.

# Appendix A

## SIM Reserved Words

Table A-1 is an alphabetic listing of SIM reserved words. You cannot use any of these words to define data names or items. Some of the reserved words contain hyphens (-); underscores (\_) can be used in place of the hyphens because SIM does not distinguish between hyphens and underscores.

**Table A-1. SIM Reserved Words**

|           |              |          |         |
|-----------|--------------|----------|---------|
| ABS       | CHARACTER    | ENDTRAN  | INSERT  |
| ADD-DAYS  | COLLATING    | EQL      | INSTEAD |
| ADD-TIME  | CONTAINS     | EQL-EQL  | INTEGER |
| AFTER     | COUNT        | EQV-GEQ  | INVERSE |
| ALL       | CURRENT      | EQV-GTR  | ISA     |
| AND       | CURRENT-DATE | EQV-LEQ  | ISIN    |
| AS        | CURRENT-TIME | EQV-LSS  | ISNOTIN |
| ASC       | DATE         | EQV-NEQ  | KANJI   |
| ASCENDING | DAY          | EXCLUDE  | LENGTH  |
| AVERAGE   | DAY-OF-WEEK  | EXCLUDES | LEQ     |
| AVG       | DELETE       | EXIST    | LEVEL   |
| BEFORE    | DESC         | EXISTS   | LIMIT   |
| BEGINTRAN | DESCENDING   | EXT      | LSS     |
| BINARY    | DISTINCT     | FALSE    | MAX     |
| BOOLEAN   | DIV          | FIRST    | MAXIMUM |
| BY        | DMLASSIGN    | FROM     | MIN     |
| CALLED    | DMLVERIFY    | GEQ      | MINIMUM |
| CANCEL    | ELAPSED-DAYS | GTR      | MINUTE  |
| CAT       | ELAPSED-TIME | HOURL    | MOD     |
| CHAR      | END          | INCLUDE  | MODIFY  |

## SIM Reserved Words

---

continued

|            |                 |             |            |
|------------|-----------------|-------------|------------|
| MONTH      | PARSE-PRIVILEGE | SOME        | TABLE      |
| MONTH-NAME | POS             | SQRT        | TIME       |
| NEQ        | PRED            | START       | TRANSITIVE |
| NO         | REAL            | STARTINSERT | TRUE       |
| NOT        | RET             | STARTMODIFY | TRUNC      |
| NUMBER     | RETRIEVE        | STRING      | WHERE      |
| OF         | ROUND           | STRUCTURE   | WITH       |
| OR         | RPT             | SUBROLE     | YEAR       |
| ORDER      | SAVE            | SUCC        |            |
| ORDERED    | SECOND          | SUM         |            |
| ORDERING   | SET             | SYMBOLIC    |            |

## Appendix B

### Syntax for Utility Commands and ODL

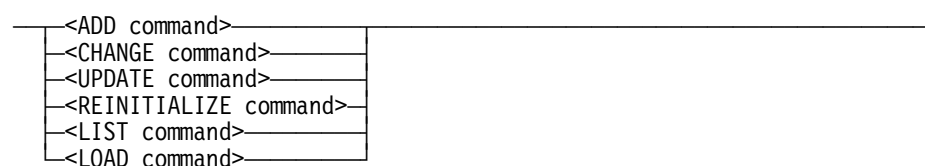
For your convenience, the railroad diagrams presented in Sections 4 through 11 of this guide are presented together in this appendix, under the following headings:

- SIM Utility Commands
- Object Definition Language (ODL)

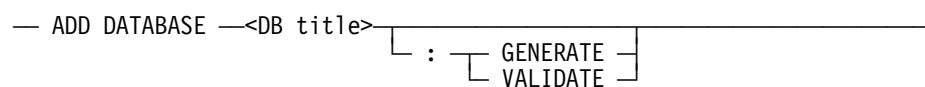
## SIM Utility Commands

The following railroad diagrams show the syntax for the SIM Utility commands relevant to SIM databases:

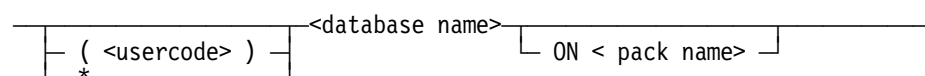
### <utility commands>



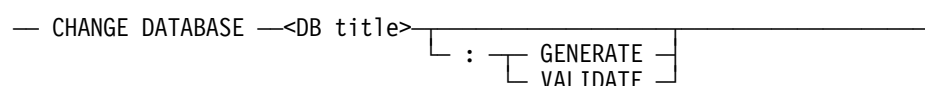
**<ADD command>**



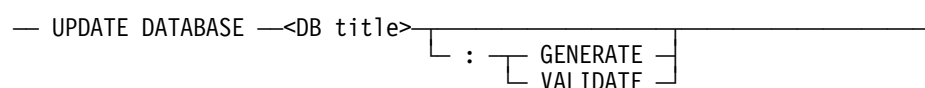
**<DB title>**



**<CHANGE command>**



**<UPDATE command>**



### <REINITIALIZE command>

— REINITIALIZE DATABASE —<DB title>—————|

### <LIST command>

— LIST DATABASE —<DB title>—|  
| : —|  
| — REMOTE —|  
| — PRINTER —|  
| — FILE <file name> —|

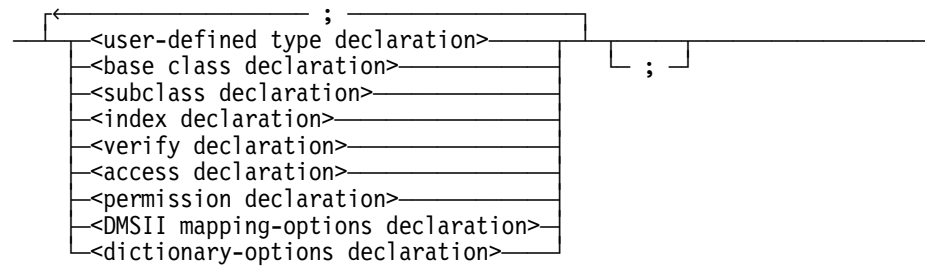
### <LOAD command>

— LOAD DATABASE —<DB title>—|  
| : — EXCLUSIVE — = —|  
| — TRUE —|  
| — FALSE —|

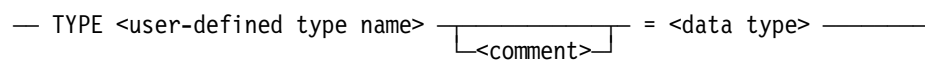
## Object Definition Language (ODL)

The following railroad diagrams show the syntax for the Object Definition Language (ODL):

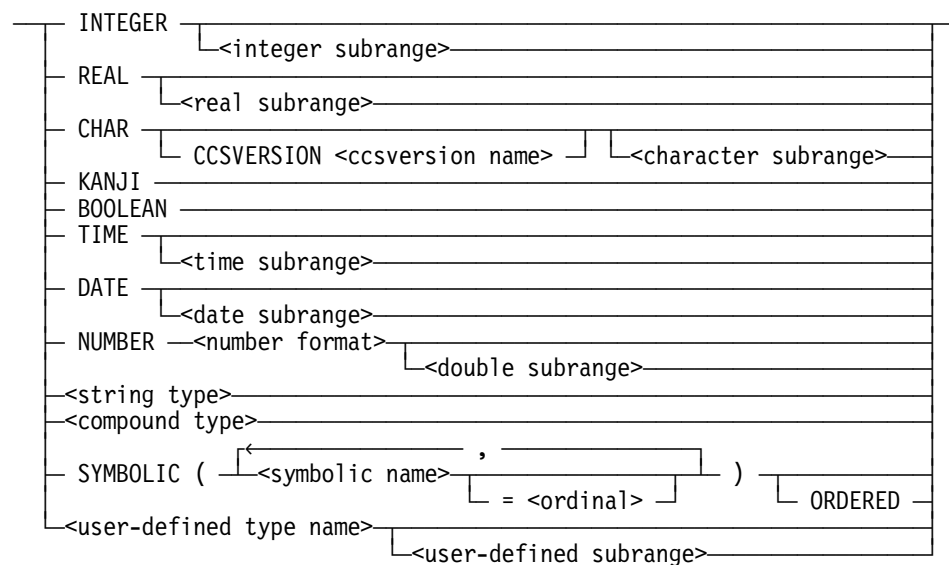
### <declarations>



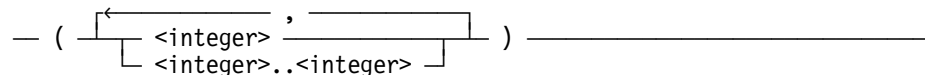
### <user-defined type declaration>



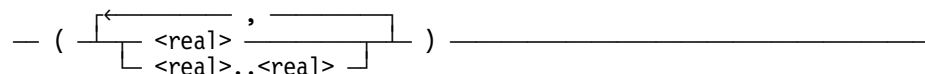
### <data type>



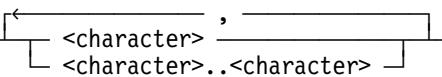
### <integer subrange>



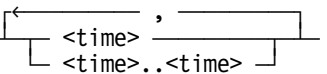
### <real subrange>



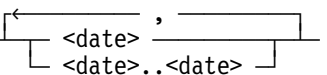
## <character subrange>

— ( [  ] ) —————


## <time subrange>

— ( [  ] ) —————

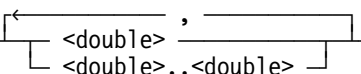
## <date subrange>

— ( [  ] ) —————

## <number format>

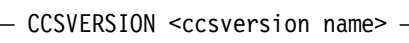
— [  <number precision> [ , <number scale> ] ] —————

## <double subrange>

— ( [  ] ) —————

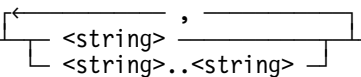
## <string type>

— STRING [ <size> ] —————→

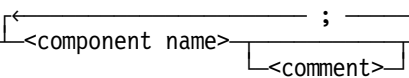
→ [ OF [ CHAR [  ] KANJI [ <user-defined type name> ] ] [ FIXED [ ] VARIABLE [ ] ] ]

→ [ <string subrange> ] —————→

## <string subrange>

— ( [  ] ) —————

## <compound type>

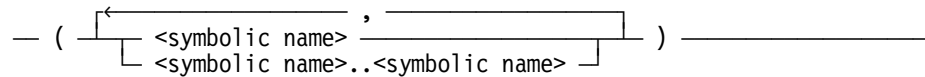
— ( [  ; [ <data type> ] [ ; ] ) —



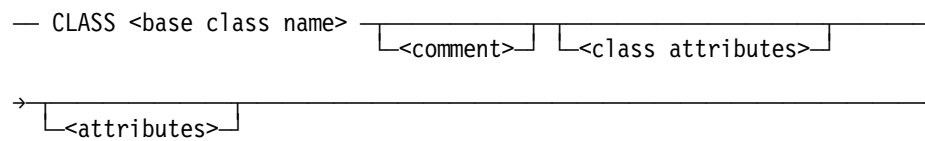
**<user-defined subrange>**



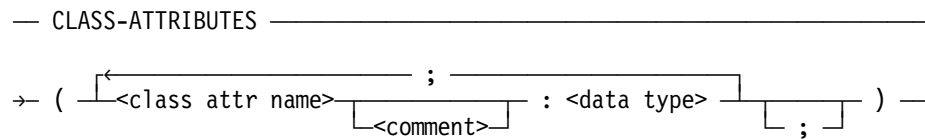
**<symbolic subrange>**



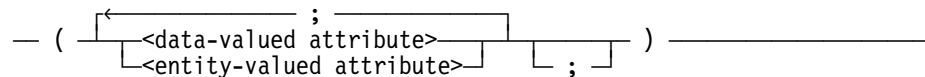
**<base class declaration>**



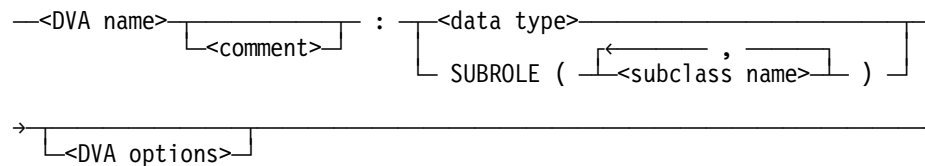
**<class attributes>**



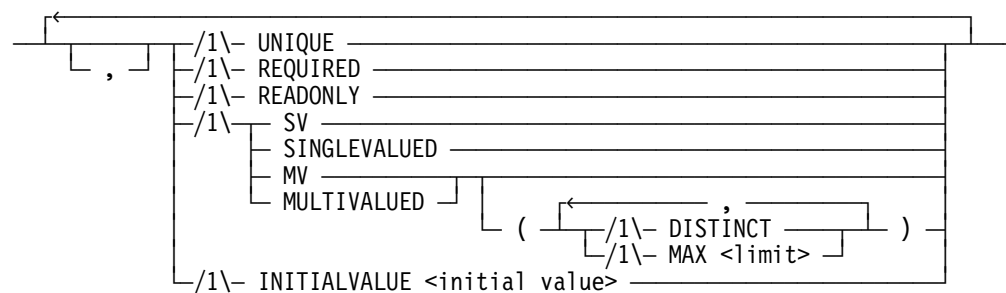
**<attributes>**



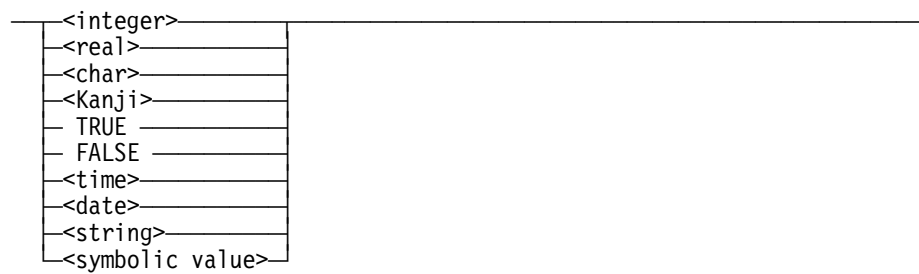
**<data-valued attribute>**



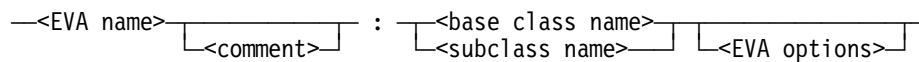
**<DVA options>**



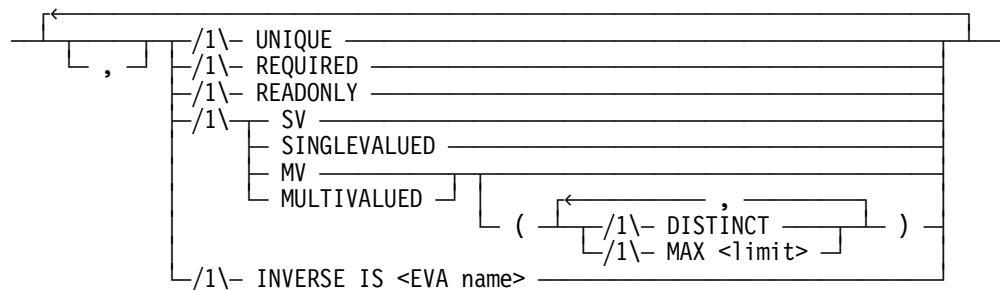
## <initial value>



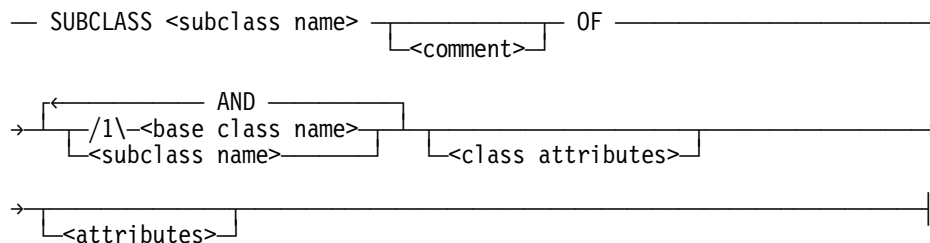
## <entity-valued attribute>



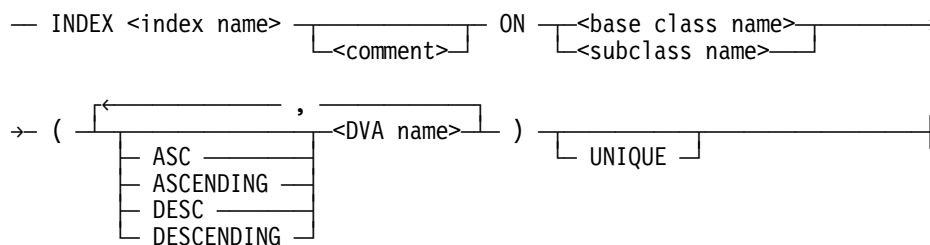
## <EVA options>



## <subclass declaration>



## <index declaration>



## <verify declaration>

— VERIFY <verify name> <comment> ON <base class name> : →  
<subclass name>  
 →<verify condition> ELSE <error message> |

## <access declaration>

— ACCESS <access name> <comment> ON <all database> <access structure> |

## <all database>

— ALLDB ALLDML |  
/1\ RETRIEVE |  
/1\ INSERT |  
/1\ DELETE |  
/1\ MODIFY |

## <access structure>

CLASS-ATTRIBUTES OF <base class name> <subclass name> |  
 → ( <attr name> , <class attr name> ) | ALLDML |  
/1\ RETRIEVE |  
/1\ INSERT |  
/1\ DELETE |  
/1\ MODIFY |  
 → WHERE <OML selection expression> |

## <permission declaration>

— PERMISSION <permission name> <comment> <access constraints> |  
 → ACCESS = <access name> |  
, |

## <access constraints>

/1\ USERCODE = ALL |  
( <usercode> ) |  
/1\ PROGRAM = ALL |  
( <program name> ) |  
/1\ ACCESSCODE = ALL |  
( <accesscode> ) |

```

— DMSII-OPTIONS ( —————→
|
|   { <global options> ; —————→ }
|   CLASS <base class name> ( ——— { <base class options> ; ——— } )
|                               | { <attr options> ; }
|   SUBCLASS <subclass name> ( ——— { <subclass options> ; ——— } )
|                               | { <attr options> ; }
|   INDEX <index name> ( <index options> ) ————— )
|   ;
— ) —————→

```

```

<
/1\-<ACCESSROUTINES specification>
/1\-<DMSUPPORT specification>
/1\-<RECOVERY specification>
/1\-<DATA RECOVERY specification>
/1\-<RECONSTRUCT specification>
/1\-<REORGANIZATION specification>
/1\-<defaults>
/1\-<options>
/1\-<parameters>
/1\-<audit trail>
/1\-<control file>
/1\- DATABASE <database name> — ( <DB physical options> )

```

```

└─ /1\- SURROGATE = TIMESTAMP
└─ <attr name>
└─ DATASET-OPTIONS ( <data set options> )

```

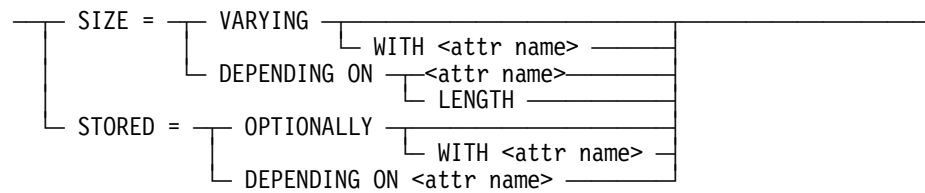
```

/1\ TYPE = STANDARD
          COMPACT
          DIRECT
          RANDOM
          UNORDERED
<data set physical option>

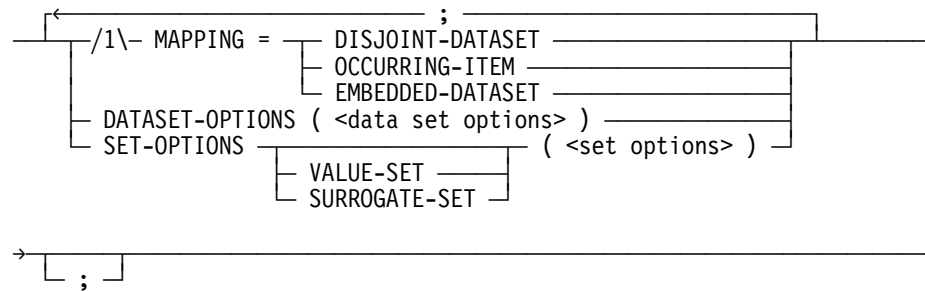
```

```
— ATTRIBUTE —<attr name>— ( —<single-valued DVA option>—  
      | —<multipvalued DVA option>—| ) —  
      | —<single-valued EVA option>—|  
      | —<multipvalued EVA option>—|
```

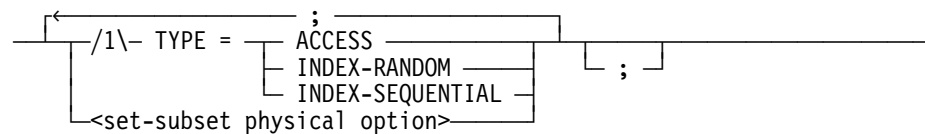
**<single-valued DVA option>**



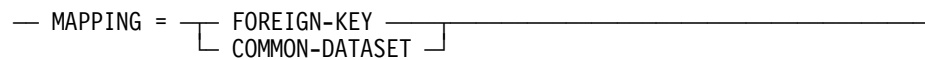
**<multivalued DVA option>**



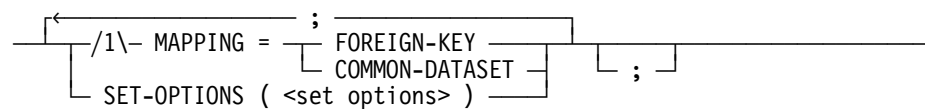
**<set options>**



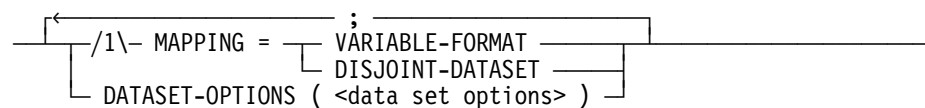
**<single-valued EVA option>**



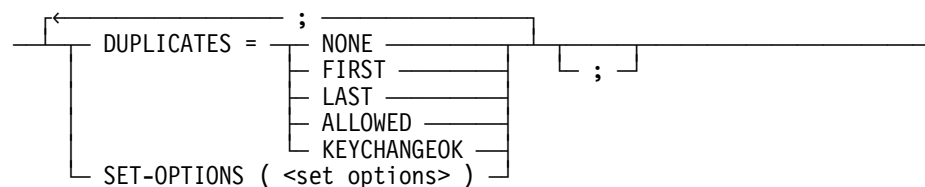
**<multivalued EVA option>**



**<subclass options>**



**<index options>**



```

— DICTIONARY-OPTIONS ( [ /1*\- DICTIONARY = <dictionary name> ]
                       [ /1*\- DIRECTORY = <directory name> ]
→ [ ; ] ) _____|

```

# Appendix C

## Understanding Railroad Diagrams

This appendix explains railroad diagrams, including the following concepts:

- Paths of a railroad diagram
- Constants and variables
- Constraints

The text describes the elements of the diagrams and provides examples.

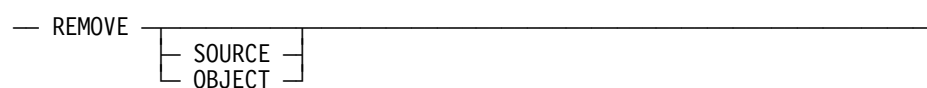
### Railroad Diagram Concepts

Railroad diagrams are diagrams that show you the standards for combining words and symbols into commands and statements. These diagrams consist of a series of paths that show the allowable structures of the command or statement.

#### Paths

Paths show the order in which the command or statement is constructed and are represented by horizontal and vertical lines. Many commands and statements have a number of options so the railroad diagram has a number of different paths you can take.

The following example has three paths:



The three paths in the previous example show the following three possible commands:

- REMOVE
- REMOVE SOURCE
- REMOVE OBJECT

A railroad diagram is as complex as a command or statement requires. Regardless of the level of complexity, all railroad diagrams are visual representations of commands and statements.

Railroad diagrams are intended to show

- Mandatory items
- User-selected items
- Order in which the items must appear
- Number of times an item can be repeated
- Necessary punctuation

Follow the railroad diagrams to understand the correct syntax for commands and statements. The diagrams serve as quick references to the commands and statements.

The following table introduces the elements of a railroad diagram:

Table C-1. Elements of a Railroad Diagram

| The diagram element . . . | Indicates an item that . . .                          |
|---------------------------|---|
| Constant                  | Must be entered in full or as a specific abbreviation |
| Variable                  | Represents data                                       |
| Constraint                | Controls progression through the diagram path         |

## Constants and Variables

A constant is an item that must be entered as it appears in the diagram, either in full or as an allowable abbreviation. If a constant is partially boldfaced, you can abbreviate the constant by

- Entering only the boldfaced letters
- Entering the boldfaced letters plus any of the remaining letters

If no part of the constant is boldfaced, the constant cannot be abbreviated.

Constants are never enclosed in angle brackets (< >) and are in uppercase letters.

A variable is an item that represents data. You can replace the variable with data that meets the requirements of the particular command or statement. When replacing a variable with data, you must follow the rules defined for the particular command or statement.

In railroad diagrams, variables are enclosed in angle brackets.

In the following example, BEGIN and END are constants, whereas <statement list> is a variable. The constant BEGIN can be abbreviated since it is partially boldfaced.

— **BEGIN** —<statement list>— END —————|



Valid abbreviations for BEGIN are

- BE
- BEG
- BEGI

### Constraints

Constraints are used in a railroad diagram to control progression through the diagram. Constraints consist of symbols and unique railroad diagram line paths. They include

- Vertical bars
- Percent signs
- Right arrows
- Required items
- User-selected items
- Loops
- Bridges

A description of each item follows.

#### Vertical Bar

The vertical bar symbol (|) represents the end of a railroad diagram and indicates the command or statement can be followed by another command or statement.

— SECONDWORD — ( —<arithmetic expression>— ) —————|

#### Percent Sign

The percent sign (%) represents the end of a railroad diagram and indicates the command or statement must be on a line by itself.

— STOP —————%

#### Right Arrow

The right arrow symbol (>)

- Is used when the railroad diagram is too long to fit on one line and must continue on the next
- Appears at the end of the first line, and again at the beginning of the next line

## Understanding Railroad Diagrams

---

— SCALERIGHT — ( —<arithmetic expression>— , —————→  
→<arithmetic expression>— ) —————|

### Required Item

A required item can be

- A constant
- A variable
- Punctuation

If the path you are following contains a required item, you must enter the item in the command or statement; the required item cannot be omitted.

A required item appears on a horizontal line as a single entry or with other items. Required items can also exist on horizontal lines within alternate paths, or nested (lower-level) diagrams.

In the following example, the word EVENT is a required constant and <identifier> is a required variable:

— EVENT —<identifier>—————|

### User-Selected Item

A user-selected item can be

- A constant
- A variable
- Punctuation

User-selected items appear one below the other in a vertical list. You can choose any one of the items from the list. If the list also contains an empty path (solid line) above the other items, none of the choices are required.

In the following railroad diagram, either the plus sign (+) or the minus sign (–) can be entered before the required variable <arithmetic expression>, or the symbols can be disregarded because the diagram also contains an empty path.

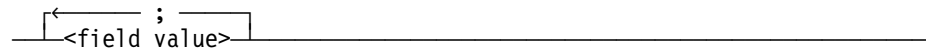
|  |   |  |
|--|---|--|
|  | + |  |
|  | – |  |

 <arithmetic expression>—————|

## Loop

A loop represents an item or a group of items that you can repeat. A loop can span all or part of a railroad diagram. It always consists of at least two horizontal lines, one below the other, connected on both sides by vertical lines. The top line is a right-to-left path that contains information about repeating the loop.

Some loops include a return character. A return character is a character—often a comma (,) or semicolon (;)—that is required before each repetition of a loop. If no return character is included, the items must be separated by one or more spaces.



## Bridge

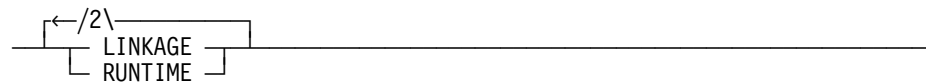
A loop can also include a bridge. A bridge is an integer enclosed in sloping lines (/ \) that

- Shows the maximum number of times the loop can be repeated
- Indicates the number of times you can cross that point in the diagram

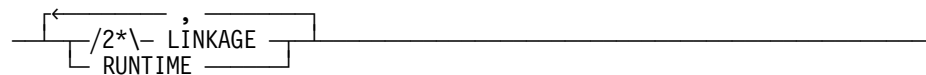
The bridge can precede both the contents of the loop and the return character (if any) on the upper line of the loop.

Not all loops have bridges. Those that do not can be repeated any number of times until all valid entries have been used.

In the first bridge example, you can enter LINKAGE or RUNTIME no more than two times. In the second bridge example, you can enter LINKAGE or RUNTIME no more than three times.



In some bridges an asterisk (\*) follows the number. The asterisk means that you must cross that point in the diagram at least once. The maximum number of times that you can cross that point is indicated by the number in the bridge.



In the previous bridge example, you must enter LINKAGE at least once but no more than twice, and you can enter RUNTIME any number of times.

### Following the Paths of a Railroad Diagram

The paths of a railroad diagram lead you through the command or statement from beginning to end. Some railroad diagrams have only one path; others have several alternate paths that provide choices in the commands or statements.

The following railroad diagram indicates only one path that requires the constant LINKAGE and the variable <linkage mnemonic>:

— LINKAGE —<linkage mnemonic>—————|

Alternate paths are provided by

- Loops
- User-selected items
- A combination of loops and user-selected items

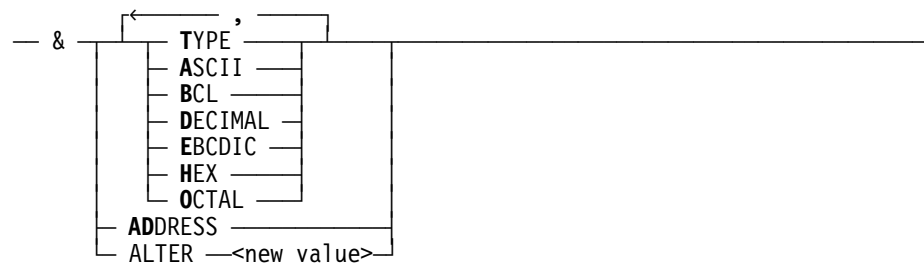
More complex railroad diagrams can consist of many alternate paths, or nested (lower-level) diagrams, that show a further level of detail.

For example, the following railroad diagram consists of a top path and two alternate paths. The top path includes

- An ampersand (&)
- Constants that are user-selected items

These constants are within a loop that can be repeated any number of times until all options have been selected.

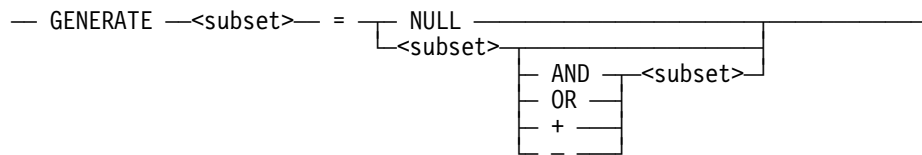
The first alternative path requires the ampersand and the required constant ADDRESS. The second alternative path requires the ampersand followed by the required constant ALTER and the required variable <new value>.





### Example 3

#### <generate statement>



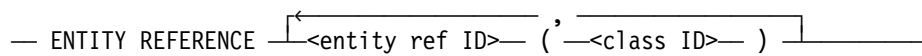
#### Sample Input

#### Explanation

|                      |   |
|----------------------|---|
| GENERATE Z = NULL    | The GENERATE constant is followed by the variable Z, an equal sign (=), and the user-selected constant NULL.  |
| GENERATE Z = X       | The GENERATE constant is followed by the variable Z, an equal sign, and the user-selected variable X.   |
| GENERATE Z = X AND B | The GENERATE constant is followed by the variable Z, an equal sign, the user-selected variable X, the AND command (from the list of user-selected items in the nested path), and a third variable, B. |
| GENERATE Z = X + B   | The GENERATE constant is followed by the variable Z, an equal sign, the user-selected variable X, the plus sign (from the list of user-selected items in the nested path), and a third variable, B.   |

### Example 4

#### <entity reference declaration>

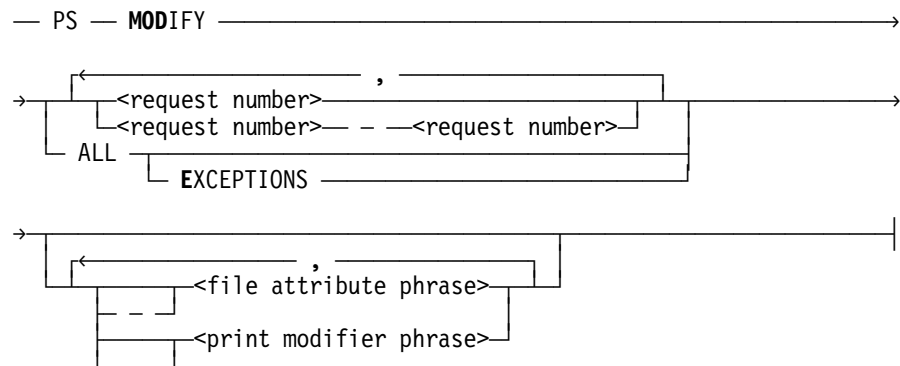


#### Sample Input

#### Explanation

|  |  |
|--|--|
| ENTITY REFERENCE ADVISOR1<br>(INSTRUCTOR)                                | The required item ENTITY REFERENCE is followed by the variable ADVISOR1 and the variable INSTRUCTOR. The parentheses are required. |
| ENTITY REFERENCE ADVISOR1<br>(INSTRUCTOR), ADVISOR2<br>(ASST_INSTRUCTOR) | Because the diagram contains a loop, the pair of variables can be repeated any number of times.                                    |

Example 5



| Sample Input                              | Explanation   |
|---|---|
| PS MODIFY 11159                           | The constants PS and MODIFY are followed by the variable 11159, which is a request number.  |
| PS MODIFY<br>11159,11160,11163            | Because the diagram contains a loop, the variable 11159 can be followed by a comma, the variable 11160, another comma, and the final variable 11163.  |
| PS MOD 11159–11161<br>DESTINATION = "LP7" | The constants PS and MODIFY are followed by the user-selected variables 11159–11161, which are request numbers, and the user-selected variable DESTINATION = "LP7", which is a file attribute phrase. Note that the constant MODIFY has been abbreviated to its minimum allowable form. |
| PS MOD ALL EXCEPTIONS                     | The constants PS and MODIFY are followed by the user-selected constants ALL and EXCEPTIONS.   |





# Index

## A

- access changes, 12-13
  - <access constraints>, 8-10, B-7
  - <access declaration>, 8-2, B-7
  - access declarations, (*See* accesses)
  - <access structure>, 8-2, B-7
- accesses, 8-1
  - basic types, 8-4
    - ALLDB access, 8-5
    - attribute access, 8-7
    - class attribute access, 8-6
  - changes, 12-13
  - examples, 8-7, 8-8
  - syntax, 8-2
- ad hoc query requirements, capturing, 2-32
- ad hoc query requirements, identifying, 2-31
  - retrieval queries, 2-32
  - update queries, 2-32
- ADD command, 11-7
  - CARD file, 11-7
  - GENERATE option, 11-8
  - syntax, 11-7
  - VALIDATE option, 11-9
- <ADD command>, 11-7, B-1
- ADD/GENERATE run, 11-9
- ADD/VALIDATE run, 11-9
- adding integrity constraints and security, 2-18
- adding semantics
  - attribute options, 2-19
  - benefits, 2-18
  - integrity and security constraints, 2-19
  - security, 2-23
  - UNIQUE indexes, 2-21
  - user-defined types, 2-20
  - verifies, 2-21
- ADDs, (*See* Advanced Data Dictionary System)
- Advanced Data Dictionary System (ADDs), 1-5
  - and the dictionary-options declaration, 10-1
  - loading a SIM database schema into ADDs, using SIM Utility, 11-19
  - used to control SIM database schemas, 1-5
    - advantages, 1-5
- <all database>, 8-2, B-7
- ALLDB access, 8-5
- allowable schema changes, (*See* changing a schema)
- application life cycle, 2-2
  - development methodologies, 2-4
  - maintenance, 2-4
  - problem statement and solution statement, 2-3
- application maintenance, 2-4
- ASERIESNATIVE, 4-7, 4-10
- <attr options>, 9-20, B-8
- attribute access, 8-7
- attribute mapping
  - EVA, 9-38
    - default, 9-39
    - many-to-many relationship, 9-46
    - one-to-many relationship, 9-43
    - one-to-one relationship, 9-41
- multivalued DVA, 9-28
  - DATASET-OPTIONS, 9-34
    - default, 9-28
    - examples, 9-33, 9-35, 9-37
    - MAPPING option, 9-30
    - options, 9-28
    - SET-OPTIONS, 9-35
    - syntax, 9-29, 9-37
- multivalued EVA, 9-40
  - syntax, 9-40
- options, 9-20
- single-valued DVA, 9-22
  - default, 9-22
  - example, 9-26
  - options, 9-23
  - SIZE options, 9-23, 9-24
  - storage compaction options, 9-23
  - STORED options, 9-23, 9-25
  - syntax, 9-23
- single-valued EVA, 9-39
  - syntax, 9-39
- attribute options, 2-19
- attribute placement, 2-11
- attribute scheme, 9-15

- attributes, 5-1, 5-10
  - data-valued attributes (DVAs), 5-10
    - DVA options, 5-12
    - example, 5-15
    - syntax, 5-11
  - distinguished from class attributes, 5-1
  - entity-valued attributes (EVAs), 5-16
    - EVA options, 5-18
    - example, 5-20
    - syntax, 5-17
  - identifying, 2-8
  - naming conventions, 2-11
  - syntax, 5-10
- <attributes>, 5-10, B-5

## B

- backup and recovery, 2-37
- base class changes, 12-5
- <base class declaration>, 5-4, B-5
- base class declarations, (*See* base classes)
- base class mapping options, 9-13
  - data set, 9-17
  - examples, 9-18, 9-19
- SURROGATE, 9-13
  - attribute scheme, 9-15
  - examples, 9-15, 9-16
  - TIMESTAMP scheme, 9-14
- <base class options>, 9-13, B-8
- base classes, 5-2, 5-4
  - changes, 12-5
  - defining, 5-4
  - example, 5-6
  - syntax, 5-4
- base type, 12-2
- bidirectional relationship, 2-12
- Boolean, 3-9
- borderline conditions and exceptions, 2-27

## C

- CANDE, (*See* Command and Edit language)
- capturing
  - ad hoc query requirements, 2-32
  - programmatic query requirements, 2-33
- CARD file, 11-20
  - with
    - ADD command, 11-7
    - CHANGE command, 11-9
    - LIST command, 11-18

- REINITIALIZE command, 11-16
- UPDATE command, 11-13
- casing of ODL syntax, 3-3
- ccsversion
  - and
    - graphic character constants, 3-9
    - graphic string constants, 3-11
    - user-defined types, 4-16
  - of CHAR data type, 4-6
  - within a STRING data type, 4-10
- CHANGE command, 11-9
  - CARD file, 11-9
  - GENERATE option, 11-10
  - syntax, 11-9
  - VALIDATE option, 11-12
- <CHANGE command>, 11-9, B-1
- CHANGE/GENERATE run, 11-10, 11-11
- CHANGE/VALIDATE run, 11-12
- changing a schema
  - access changes, 12-13
  - allowed changes, 12-1
  - base class changes, 12-5
  - CHANGE command, 11-9
  - syntax, 11-9
  - changes not allowed, 12-1
  - circumventing restrictions, 12-16
  - class attribute changes, 12-6
  - data-valued attribute (DVA) changes, 12-8
  - dictionary option changes, 12-16
  - DMSII mapping option changes, 12-14
  - effects on queries, 12-3
  - entity-valued attribute (EVA)
    - changes, 12-11
  - index changes, 12-12
  - long-term requirements, 2-38
  - subclass changes, 12-6
  - user-defined type changes, 12-4
  - verify changes, 12-13
- CHAR data type, 4-6
  - ccsversion, 4-6
- character, 3-8
  - graphic format, 3-8
  - hexadecimal format, 3-9
- character subrange, 4-6
- <character subrange>, 4-6, B-4
- circumventing schema change
  - restrictions, 12-16
- class attribute access, 8-6
- class attribute changes, 12-6
- class attribute mapping, 9-21
- class attributes, 5-1, 5-8
  - changes, 12-6
  - distinguished from attributes, 2-11, 5-1

- example, 5-9
- syntax, 5-8
- <class attributes>, 5-8, B-5
- classes, 5-1
  - attributes, 5-1, 5-10
    - data-valued, 5-1
    - data-valued attributes (DVAs), 5-10
    - entity-valued, 5-1
    - entity-valued attributes (EVAs), 5-16
  - identifying, 2-8
  - syntax, 5-10
- base classes, 5-2
  - example, 5-6
  - syntax, 5-4
- class attributes, 5-1, 5-8
  - example, 5-9
  - syntax, 5-8
- concepts, 5-2
  - entity relationships, 5-2
  - generalization hierarchies, 5-2
- data-valued attributes (DVAs), 5-1, 5-10
  - example, 5-15
  - syntax, 5-11
- entity relationships, 5-2
- entity-valued attributes (EVAs), 5-1, 5-16
  - example, 5-20
  - syntax, 5-17
- generalization hierarchies, 5-2
  - forming hierarchies, 2-14
  - least common ancestor (LCA), 5-3
  - rules, 5-3
- identifying, 2-8
- multiple immediate subclasses, 5-3
- multiple immediate superclasses, 5-3
- naming conventions, 2-11
- subclasses, 5-6
  - example, 5-8
  - syntax, 5-6
- superclasses, 5-2
- coded character set version, (*See* ccsversion)
- Command and Edit language (CANDE), 1-4, 3-3
  - EXECUTE command, 11-4
  - RUN command, 11-4
- commands, SIM Utility, 11-7
  - ADD, 11-7
    - CARD file, 11-7
    - GENERATE option, 11-8
    - syntax, 11-7
    - VALIDATE option, 11-9
  - CHANGE, 11-9
    - CARD file, 11-9
    - GENERATE option, 11-10
    - syntax, 11-9
    - VALIDATE option, 11-12
  - LIST, 11-18
    - CARD file, 11-18
    - PRINTER option, 11-5, 11-18
    - REMOTE option, 11-5, 11-18
    - syntax, 11-18
  - LOAD, 11-19
    - CARD file, 11-19
    - EXCLUSIVE = TRUE option, 11-19
    - syntax, 11-19
  - REINITIALIZE, 11-16
    - CARD file, 11-16
    - syntax, 11-16
  - UPDATE, 11-13
    - CARD file, 11-13, 11-15
    - copying and updating a database, 11-16
    - GENERATE option, 11-14
    - syntax, 11-13
    - updating software levels, 11-15
    - VALIDATE option, 11-15
- comments in a schema
  - escape comments, 3-6
  - official comments, 2-10, 3-6
- common data set mapping
  - one-to-many relationship, 9-45
  - one-to-one relationship, 9-42
- competing views, 2-30
- <compound type>, 4-3, B-4
- compounds versus classes and EVAs, 2-29
- conflicting views, 2-30
- constant data values, 3-7
  - Boolean, 3-9
  - character, 3-8
    - graphic format, 3-8
    - hexadecimal format, 3-9
  - date, 3-10
  - double, 3-11
  - integer, 3-7
  - Kanji, 3-9
  - real, 3-8
  - string, 3-11
    - 16-bit character strings, 3-12
    - graphic format, 3-11, 3-12
    - hexadecimal format, 3-12
    - implicit concatenation, 3-12
  - time, 3-10
- constructs, ODL, 1-6
- copying a database, 11-13
- creating a new SIM database, 11-7
- creating and managing a schema file, 3-3
- critical examples, 2-15

## D

- DAG (directed acyclic graph), 5-3
- DASDL file, 11-20
- data definition language (DDL), 1-3
- Data Management System II (DMSII) mapping
  - options, (*See* DMSII mapping options)
- data manipulation language (DML), 1-3
- data models, 1-2
- data set options
  - for a
    - base class, 9-17
    - subclass, 9-51
- <data set options>, 9-17, B-8
- <data subrange>, 4-8
- <data type>, 4-2, B-3
- data types
  - and formats, 2-34
  - constructor, 4-2
  - primitive, 4-2
    - CHAR, 4-6
- database design, 2-6
  - overview, 2-6
  - phases
    - interface design, 2-31
    - logical design, 2-7
    - physical design, 2-35
- database design guidelines, (*See* design guidelines)
- database development methodologies
  - entity-relationship (E-R) modeling, 2-4
  - SIM data model, 2-5
- database management systems, 1-1
- database schema development, 1-4
  - Advanced Data Dictionary System (ADDS), 1-5
  - direct ODL approach, 1-4
  - SIM Design Assistant (SDA), 1-4
- database schema file
  - means and media for creating, 3-3
  - processing, 3-3
- database schemas, (*See* schema)
- database types, 11-1
  - generated, 11-2
  - validated, 11-2
  - viewed, 11-3
- databases, 1-1
- data-valued attribute (DVA) changes, 12-8
- <data-valued attribute>, 5-11, B-5
- data-valued attributes (DVAs), 5-1, 5-10
  - changes, 12-8
  - DVA options, 5-12
  - DISTINCT, 5-14
  - INITIALVALUE, 5-15
  - MAX <limit>, 5-14
  - MV or MULTIVALUED, 5-14
  - READONLY, 5-13
  - REQUIRED, 5-13
  - SV or SINGLEVALUED, 5-14
  - UNIQUE, 5-13
  - subrole DVA, 5-10
  - syntax, 5-11
- date, 3-10
- <date subrange>, B-4
- <DB title>, 11-7, B-1
- DBMS, 1-1
- DDL, 1-3
- DDLRESULTS file, 11-6, 11-21
  - examining error and warning messages, 11-21
- deciding between DVA or EVA, 2-12
- declarations, (*See* ODL declarations)
- <declarations>, 3-1, B-3
- design alternatives and special situations, 2-29
  - compounds versus classes and EVAs, 2-29
  - conflicting views, 2-30
  - historic data, 2-30
  - multiple inheritance, 2-29
  - null attributes versus new subclasses, 2-29
  - transitional constraints, 2-29
- Design Assistant, (*See* SIM Design Assistant)
- design guidelines, 2-1
  - application life cycle, 2-2
  - attribute options, 2-19
  - attribute placement, 2-11
  - attribute types, 2-12
  - backup and recovery, 2-37
  - capturing
    - ad hoc query requirements, 2-32
    - programmatic query requirements, 2-33
  - compounds versus classes and EVAs, 2-29
  - conflicting views, 2-30
  - critical examples, 2-15
  - data types and formats, 2-34
  - deciding between DVA or EVA, 2-12
  - EVA/attribute ratios, 2-24
  - forming hierarchies, 2-14
  - fringe conditions, 2-27
  - hierarchy relationships between
    - classes, 2-15
  - historic data, 2-30
  - information = data + rules, 2-25
  - initial considerations, 2-1
  - integrity and security constraints, 2-19
  - interface design phase, 2-31

- evaluating and refining, 2-34
- identifying ad hoc query requirements, 2-31
- identifying programmatic query requirements, 2-33
- steps, 2-31
- isolated islands of data, 2-29
- logical design phase, 2-7
  - adding semantics, 2-18
  - evaluating and refining, 2-24
  - generalizing and specializing, 2-14
  - identifying things and their properties, 2-7
  - steps, 2-7
- many-to-many relationships, 2-28
- matching requirements to the schema, 2-24
- multiple inheritance, 2-29
- naming conventions for class and attribute names, 2-11
- nonrequired attributes, 2-27
- null attributes versus new subclasses, 2-29
- physical design phase, 2-35
  - evaluating and refining, 2-39
  - identifying capacity requirements, 2-35
  - identifying operational requirements, 2-37
  - identifying performance requirements, 2-36
  - steps, 2-35
- prototyping, 2-5
- redundant data, 2-16
- relationships, 2-12
- retrieval queries, 2-32
- role relationships, 2-15
  - rules, 2-15
- schema control, 2-38
- schema evolution, 2-38
- security, 2-23
- SIM data model beyond databases, 2-1, 2-40
- SIM database design, 2-1
- tradeoffs and special cases, 2-29
- transitional constraints, 2-29
- UNIQUE indexes, 2-21
- unnecessary classes, 2-26
- update queries, 2-32, 2-33
- user-defined types, 2-20
- verifies, 2-21
- development methodologies, 2-4
  - database development, 2-4
  - program development, 2-4
- dictionary option changes, 12-16
- dictionary options
  - advantages, 10-2
  - changes, 12-16
  - effect on database, 10-2, 10-3
    - when first used through an ADD or CHANGE command, 10-3
    - when modified with a CHANGE command, 10-3
  - example, 10-4
  - syntax, 10-1
- <dictionary-options declaration>, 10-1, B-10
- dictionary-options declarations, (*See* dictionary options)
- directed acyclic graph (DAG), 5-3
- DMSII mapping option changes, 12-14
- DMSII mapping options, 9-1
  - attribute, 9-20
    - EVA mapping, 9-38
    - multivalued DVA, 9-28
    - multivalued EVA, 9-40
    - single-valued DVA, 9-23
    - single-valued EVA, 9-39
  - syntax, 9-20
- base class, 9-13
  - data set options, 9-17
  - SURROGATE, 9-13
  - syntax, 9-13, 9-17
- changes, 12-14
- global, 9-3
  - example, 9-12
  - guard files with SIM databases, 9-9
  - performance considerations, 9-7
  - restrictions and ramifications, 9-5
  - syntax, 9-3
- index, 9-53
  - DUPLICATES, 9-54
  - SET-OPTIONS, 9-55
  - syntax, 9-53
- monitoring and analysis tools, 2-39, 9-2
- multivalued DVA, 9-28
  - syntax, 9-29, 9-37
- multivalued EVA, 9-40
  - syntax, 9-40
- single-valued DVA, 9-23
  - syntax, 9-23
- single-valued EVA, 9-39
  - syntax, 9-39
- subclass, 9-47
  - DATASET-OPTIONS, 9-51
  - MAPPING, 9-47
  - syntax, 9-47
- syntax, 9-2
  - when to declare, 9-1
- <DMSII mapping-options declaration>, 9-2, B-8

- double, 3-11
- <double subrange>, 4-9, B-4
- DVA changes, 12-8
- DVA mapping
  - multivalued DVA, 9-28
    - DATASET-OPTIONS, 9-34
    - default, 9-28
    - examples, 9-33, 9-35, 9-37
    - MAPPING option, 9-30
    - options, 9-28
    - SET-OPTIONS, 9-35
    - syntax, 9-29, 9-37
  - single-valued DVA, 9-22
    - default, 9-22
    - example, 9-26
    - options, 9-23
    - SIZE options, 9-24
    - storage compaction options, 9-23
    - STORED options, 9-25
    - syntax, 9-23
- DVA options, 5-12
  - DISTINCT, 5-14
  - INITIALVALUE, 5-15
  - MV or MULTIVALUED, 5-14
  - READONLY, 5-13
  - REQUIRED, 5-13
  - SV or SINGLEVALUED, 5-14
  - UNIQUE, 5-13
- <DVA options>, 5-11, B-5
- DVAs, (*See* data-valued attributes)

## E

- Editor program, 1-4, 3-3
  - used with DDLRESULTS file, 11-21
- effects of schema changes on queries, 12-3
- entities, 1-3, 5-1
  - surrogate value, 5-1
- entity relationships, 5-16
- entity-relationship (E-R) modeling, 2-4
- entity-valued attribute (EVA) changes, 12-11
- <entity-valued attribute>, 5-17, B-6
- entity-valued attributes (EVAs), 5-1, 5-16
  - changes, 12-11
  - classes and EVAs versus compounds, 2-29, 4-13
- EVA options, 5-18
  - DISTINCT, 5-19
  - INVERSE IS <EVA name>, 5-20
  - MAX <limit>, 5-20
  - MV or MULTIVALUED, 5-19

- READONLY, 5-19
- REQUIRED, 5-19
- SV or SINGLEVALUED, 5-19
- UNIQUE, 5-19
- example, 5-20
- implicit EVAs, 5-16
- inverse EVAs, 5-17
- perspective class, 5-16
- referential integrity for relationships, 5-17
- reflexive relationships, 5-17
- relationships between
  - declared EVA and implicit EVA, 5-16
  - explicitly declared EVAs, 5-16
- self-reflexive, 5-17
- syntax, 5-17
- target class, 5-16
- E-R modeling, 2-4
- escape comments, 3-6
- EVA changes, 12-11
- EVA mapping, 9-38
  - default, 9-39
  - many-to-many relationship, 9-46
  - multivalued EVA mapping options, 9-40
    - syntax, 9-40
  - one-to-many relationship, 9-43
    - common data set, 9-45
    - foreign-key, 9-43
  - one-to-one relationship, 9-41
    - common data set, 9-42
    - foreign-key, 9-41
  - single-valued EVA mapping options, 9-39
    - syntax, 9-39
- EVA options, 5-18
  - DISTINCT, 5-19
  - INVERSE IS <EVA name>, 5-20
  - MAX <limit>, 5-20
  - READONLY, 5-19
  - REQUIRED, 5-19
  - SV or SINGLEVALUED, 5-19
  - UNIQUE, 5-19
- <EVA options>, 5-18, B-6
- EVA/attribute ratios, 2-24
- EVAs, (*See* entity-valued attributes)
- example database schema development, 1-7
- examples of ODL constructs
  - access, 8-7, 8-8
  - attribute mapping options
    - multivalued DVA, 9-33, 9-35, 9-37
    - single-valued DVA, 9-26
  - base class, 5-6
  - base class mapping options
    - data set, 9-18
    - SURROGATE, 9-15

- class attributes, 5-9
- data-valued attributes (DVAs), 5-15
- dictionary options, 10-4
- entity-valued attributes (EVAs), 5-20
- global mapping options, 9-12
- index, 6-3, 6-5
- index mapping options, 9-56, 9-57
- permission, 8-12
- subclass, 5-8
- subclass mapping options, 9-48, 9-50, 9-52
- user-defined type, 4-19
- verify, 7-5
- exceptions and borderline conditions, 2-27
- EXCLUSIVE = TRUE option in LOAD
  - command, 11-19
- executing the SIM Utility, 11-4
  - batch execution, 11-5
  - conflicting executions, 11-6
  - input files, 11-4
  - output files, 11-4
  - program parameter, 11-4
  - remote execution, 11-4
  - restarts, 11-6
  - status file, 11-6
- extracting a SIM database, 11-18

## F

- files, SIM Utility, 11-20
  - CARD, 11-20
  - DASDL, 11-20
  - DDLRESULTS, 11-21
  - LINE, 11-22
- fine-tuning a schema, 2-24
- foreign-key mapping
  - one-to-many relationship, 9-43
  - one-to-one relationship, 9-41
- forming hierarchies, 2-14
- fringe conditions, 2-27

## G

- garbage collection, 2-39
- generalization hierarchies, 2-14, 5-2
  - forming hierarchies, 2-14
  - rules, 5-3
- generalizing and specializing, 2-14
  - critical examples, 2-15
  - forming hierarchies, 2-14
  - redundant data, 2-16

- role relationships
  - rules, 2-15
- GENERATE option
  - for
    - ADD command, 11-8
    - CHANGE command, 11-10
    - UPDATE command, 11-14
- GENERATE run, 11-2
- generated databases, 11-2
- global mapping options, 9-3
  - example of use in declaration, 9-12
  - guard files, 9-9
  - performance considerations, 9-7
  - restrictions and ramifications, 9-5
    - audit trail attributes, 9-6
    - code file titles, 9-5
    - control files, 9-6
    - database physical options, 9-7
    - defaults, 9-5
    - options, 9-5
    - parameters, 9-6
  - SECURITYGUARD specifications, 9-9
- <global options>, 9-3, B-8
- graphic character format, 3-8
- graphic string format, 3-11
- guard files with SIM databases, 9-9
- guidelines, SIM database design, (*See* design guidelines)

## H

- hexadecimal character format, 3-9
- hexadecimal string format, 3-12
- hierarchies, forming, 2-14
- historic data, 2-30
- hyphens and underscores, 3-3, A-1

## I

- identifiers, (*See* SIM identifiers)
- identifying ad hoc query requirements, 2-31
  - capturing queries, 2-32
  - retrieval queries, 2-32
  - update queries, 2-32
- identifying attributes, 2-8
- identifying capacity requirements, 2-35
- identifying classes, 2-8
- identifying operational requirements, 2-37
  - backup and recovery, 2-37
  - schema control, 2-38

- schema evolution, 2-38
- identifying performance requirements, 2-36
- identifying programmatic query requirements, 2-33
  - capturing queries, 2-33
  - data types and formats, 2-34
  - update queries, 2-33
- identifying things and their properties, 2-7
  - attribute placement, 2-11
  - attribute types, 2-12
  - attributes, 2-8
  - class attributes, 2-11
  - classes, 2-8
  - naming conventions, 2-11
  - relationships, 2-12
- immediate type, 12-2
- implicit EVAs, 5-16
- implicit string concatenation, 3-12
- index changes, 12-12
- <index declaration>, 6-2, B-6
- index declarations, (*See* indexes)
- index mapping options, 9-53
  - DUPLICATES, 9-54
  - SET-OPTIONS, 9-55
  - examples, 9-56, 9-57
- <index options>, 9-53, B-9
- indexes, 6-1
  - changes, 12-12
  - equality searches, 6-4
  - examples, 6-3, 6-5
  - guidelines for using, 6-4
  - inequality searches, 6-4
  - key, 6-1
  - multiple key attributes, 6-5
  - name, 6-1
  - parts, 6-1
  - purposes, 6-1
  - restrictions
    - on all indexes, 6-3
    - on UNIQUE indexes, 6-4
  - SIM query optimizer, 6-1, 6-4
  - syntax, 6-2
  - target, 6-1
  - UNIQUE indexes, 2-21
- InfoExec system, 1-3
- information = data + rules, 2-25
- <initial value>, 5-11, B-6
- integer, 3-7
- integer subrange, 4-5
  - <integer subrange>, 4-5, B-3
- integrity constraints and security, adding, 2-18
- interface design phase, 2-31
  - evaluating and refining, 2-34

- identifying ad hoc query requirements, 2-31
  - capturing queries, 2-32
  - retrieval queries, 2-32
  - update queries, 2-32
- identifying programmatic query requirements, 2-33
  - capturing queries, 2-33
  - data types and formats, 2-34
  - update queries, 2-33
- steps, 2-31
  1. Identify ad-hoc query requirements, 2-31
  2. Identify programmatic query requirements, 2-33
  3. Evaluate and refine, 2-34
- introduction to ODL, 1-1
  - database management systems, 1-1
  - database schema development, 1-4
    - Advanced Data Dictionary System (ADDs), 1-5
    - direct ODL approach, 1-4
    - SIM Design Assistant (SDA), 1-4
  - database schemas, 1-3
  - databases, 1-1
  - entities and objects, 1-3
  - example database schema development, 1-7
  - InfoExec system, 1-3
  - ODL declarations, 1-6
  - Semantic Information Manager (SIM), 1-2
- inverse EVAs, 5-17
- isolated islands of data, 2-29

## K

- Kanji, 3-9

## L

- language conventions, ODL, 3-4
- LCA (least common ancestor), 5-3
- least common ancestor (LCA), 5-3
- LIMIT option, 11-23
  - syntax, 11-23
- LINE file, 11-22
- LIST command, 11-18
  - CARD file, 11-18
  - performed on a viewed database, 11-18
  - PRINTER option, 11-5, 11-18
  - REMOTE option, 11-5, 11-18
  - syntax, 11-18



- <LIST command>, 11-18, B-2
- LIST option, 11-23
  - syntax, 11-23
- listing a SIM database schema, 11-18
- LOAD command, 11-19
  - CARD file, 11-19
  - EXCLUSIVE = TRUE option, 11-19
  - syntax, 11-19
- <LOAD command>, 11-19, B-2
- loading a SIM database schema into ADDS,
  - using SIM Utility, 11-19
- locking implications in using verifies, 7-4
- logical design phase, 2-7
  - adding semantics, 2-18
    - attribute options, 2-19
    - integrity and security constraints, 2-19
    - security, 2-23
    - UNIQUE indexes, 2-21
    - user-defined types, 2-20
    - verifies, 2-21
  - deciding between DVA or EVA, 2-12
  - evaluating and refining, 2-24
    - EVA/attribute ratios, 2-24
    - fringe conditions, 2-27
    - information = data + rules, 2-25
    - isolated islands of data, 2-29
    - many-to-many relationships, 2-28
    - matching requirements to the
      - schema, 2-24
    - nonrequired attributes, 2-27
    - tradeoffs and special cases, 2-29
    - unnecessary classes, 2-26
  - generalizing and specializing, 2-14
    - critical examples, 2-15
    - forming hierarchies, 2-14
    - redundant data, 2-16
    - role relationships, 2-15
  - identifying attributes and their properties
    - attribute types, 2-12
  - identifying things and their properties, 2-7
    - attribute placement, 2-11
    - attributes, 2-8
    - class attributes, 2-11
    - classes, 2-8
    - naming conventions, 2-11
    - relationships, 2-12
  - steps, 2-7
    1. Identify things and their properties, 2-7
    2. Generalize and specialize, 2-14
    3. Add semantics, 2-19
    4. Evaluate and refine, 2-24

## M

- maintenance, application, 2-4
- many-to-many relationship mapping, 9-46
- many-to-many relationships, 2-28
- mapping options, DMSII, (*See* DMSII mapping options)
- MARC RUN command, 11-4
- matching requirements to the schema, 2-24
- Menu-Assisted Resource Control (MARC)
  - RUN command, 11-4
- monitoring and analysis tools, DMSII, 2-39, 9-2
- multiple immediate subclasses, 5-3
- multiple immediate superclasses, 5-3
- multiple inheritance, 2-29
- multivalued DVA mapping, 9-28
  - default, 9-28
  - examples, 9-33, 9-35, 9-37
  - options, 9-28
    - DATASET-OPTIONS, 9-34
    - MAPPING, 9-30
    - SET-OPTIONS, 9-35
  - syntax, 9-29, 9-37
- <multivalued DVA option>, 9-29, B-9
- multivalued EVA mapping options, 9-40
  - syntax, 9-40
- <multivalued EVA option>, 9-40, B-9
- multivalued subrole DVAs, 5-14, 9-28

## N

- naming conventions for class and attribute
  - names, 2-11
- native ccsversion, 4-10
- nonrequired attributes, 2-27
- null attributes versus new subclasses, 2-29
- <number format>, 4-8, B-4

## O

- Object Definition Language (ODL)
  - language conventions, 3-4
  - overview, 1-1
  - syntax rules, 3-3
- Object Manipulation Language (OML), 1-1
  - defining OML operations in an access, 8-1
  - global selection expression
    - in a verify, 7-3, 7-4
    - in an access, 8-4

- library interface, 11-2, 11-3
- PRED and SUCC functions, 4-15
- queries, 2-5, 2-32, 2-34, 2-39
- statements, 1-3, 5-9, 5-10, 5-19
  - used in circumventing schema change restrictions, 12-16
- object-oriented databases (OODBs), 1-2
- objects, 1-3
- OCM, 2-39
- ODL, (*See* Object Definition Language)
- ODL constructs, 1-6
- ODL declarations
  - access, 8-1
  - base class, 5-4
  - constructs, 1-6
  - dictionary-options, 10-1
  - DMSII mapping-options, 9-1
  - index, 6-1
  - kinds of, 1-6, 3-1
  - parts of, 3-4
  - permission, 8-9
  - subclass, 5-6
  - syntax for specifying, 3-1
  - verify, 7-1
- ODL syntax, (*See* syntax, ODL)
- official comments, 2-10, 3-6
- OML, (*See* Object Manipulation Language)
- one-to-many relationship mapping, 9-43
- one-to-one relationship mapping, 9-41
- OODBs (object-oriented databases), 1-2
- Operations Control Manager (OCM), 2-39
- options, SIM Utility, 11-23
  - LIMIT, 11-23
    - syntax, 11-23
  - LIST, 11-23
    - syntax, 11-23
  - WARNSUPR, 11-24
    - syntax, 11-24
- ORGANIZATION database
  - casing of ODL syntax, 3-17
  - description, 3-14
  - relationships between the classes, 3-14
  - schema, 3-17

## P

- parts of a declaration, 3-4
- <permission declaration>, 8-10, B-7
- permission declarations, (*See* permissions)
- permissions, 8-1, 8-9
  - examples, 8-12

- syntax, 8-10
- persistent data, 1-1, 2-7
- perspective class, 5-16
- physical design phase, 2-35
  - evaluating and refining, 2-39
  - identifying capacity requirements, 2-35
  - identifying operational requirements, 2-37
    - backup and recovery, 2-37
    - schema control, 2-38
    - schema evolution, 2-38
  - identifying performance requirements, 2-36
- steps, 2-35
  1. Identify capacity requirements, 2-35
  2. Identify performance requirements, 2-36
  3. Identify operational requirements, 2-37
  4. Evaluate and refine, 2-39
- precision, 2-34
- primitive types
  - CHAR, 4-6
- program development methodologies
  - object-oriented analysis and design, 2-4
  - structured analysis and design, 2-4
- programmatic query requirements,
  - capturing, 2-33
- programmatic query requirements,
  - identifying, 2-33
  - data types and formats, 2-34
- programming query requirements, identifying
  - update queries, 2-33
- prototyping, 2-5
  - and SIM, 2-5

## Q

- queries, effects of schema changes on, 12-3

## R

- railroad diagrams
  - ODL, (*See* syntax, ODL)
  - SIM Utility commands, (*See* syntax, SIM Utility commands)
- railroad diagrams, explanation of, C-1
- real, 3-8
- real subrange, 4-5
- <real subrange>, 4-5, B-3
- redundant data, 2-16
- referential integrity for relationships, 5-17
- reflexive relationships, 5-17

REINITIALIZE command, 11-16  
     CARD file, 11-16  
     difference from DMUTILITY INITIALIZE  
         statement, 11-17  
     functionality, 11-17  
     syntax, 11-16  
 <REINITIALIZE command>, 11-16, B-2  
 reinitializing a database, 11-16  
     that contains verifies, 11-17  
 relationship attributes, 2-12  
 relationship instances, 2-14  
 relationships between the classes in  
     ORGANIZATION database, 3-14  
 reserved words, A-1  
 retrieval queries, 2-32  
 role relationship rules, 2-15  
 RUN command, 11-4

## S

schema, 1-3  
     changes, (*See* changing a schema)  
     creating, 1-4  
         ADDS, 1-5  
         SDA, 1-4  
         text file, 1-4  
     development, 1-4  
         Advanced Data Dictionary System  
             (ADDS), 1-5  
         direct ODL approach, 1-4  
         example, 1-7  
         SIM Design Assistant (SDA), 1-4  
     loading into ADDS by using the SIM  
         Utility, 11-19  
     modifying, 1-4  
         ADDS, 1-5  
         SDA, 1-4  
         text file, 1-4  
     schema changes, (*See* changing a schema)  
     schema control, 2-38  
     schema evolution, 2-38  
     schema file  
         means and media for creating, 3-3  
         processing, 3-3  
     SDA (SIM Design Assistant), 1-4  
     security, SIM database, 8-1  
         access declarations, 8-1  
         logical, 8-1  
         permission declarations, 8-9  
         physical, 8-1  
     SECURITYGUARD specifications, 9-9

self-reflexive EVAs, 5-17  
 semantic data model, 2-5  
 Semantic Information Manager (SIM), 1-2  
 semantics, adding  
     attribute options, 2-19  
     benefits, 2-18  
     integrity and security constraints, 2-19  
     security, 2-23  
     UNIQUE indexes, 2-21  
     user-defined types, 2-20  
     verifies, 2-21  
 semicolons in a schema, 3-3  
 <set options>, 9-37, B-9  
 siblings, 2-14  
 SIM (Semantic Information Manager), 1-2  
 SIM data model, 2-5  
     beyond databases, 2-40  
     design uses other than for databases, 2-40  
 SIM database design, 2-6  
     overview, 2-6  
     phases, 2-6  
         interface design, 2-31  
         logical design, 2-7  
         physical design, 2-35  
 SIM database design guidelines, (*See* design  
     guidelines)  
 SIM database development, 2-5  
 SIM database names, 11-8  
 SIM database schema changes, (*See* changing  
     a schema)  
 SIM database security, (*See* security, SIM  
     database)  
 SIM Design Assistant (SDA), 1-4  
 SIM directory, 10-2  
 SIM identifiers, 3-3  
     rules for using, 3-5  
 SIM reserved words, A-1  
 SIM Utility, 11-1  
     ADD command, 11-7  
         CARD file, 11-7  
         GENERATE option, 11-8  
         syntax, 11-7  
         VALIDATE option, 11-9  
     ADD/GENERATE run, 11-9  
     ADD/VALIDATE run, 11-9  
     CARD file, 11-20  
     CHANGE command, 11-9  
         CARD file, 11-9  
         GENERATE option, 11-10  
         syntax, 11-9  
         VALIDATE option, 11-12  
     CHANGE/GENERATE run, 11-10, 11-11  
     CHANGE/VALIDATE run, 11-12

- commands, 11-7
  - ADD, 11-7
  - CHANGE, 11-9
  - LIST, 11-18
  - LOAD, 11-19
  - REINITIALIZE, 11-16
  - UPDATE, 11-13
- conflicting executions, 11-6
- DASDL file, 11-20
- database types, 11-1
  - generated, 11-2
  - validated, 11-2
  - viewed, 11-3
- DDLRESULTS file, 11-6, 11-21
  - examining error and warning messages, 11-21
- executing, 11-4
  - batch execution, 11-5
  - conflicting executions, 11-6
  - input files, 11-4
  - output files, 11-4
  - program parameter, 11-4
  - remote execution, 11-4
  - restarts, 11-6
  - status file, 11-6
- files
  - CARD, 11-20
  - DASDL, 11-20
  - DDLRESULTS, 11-6, 11-21
  - LINE, 11-22
- generated databases, 11-2
- LINE file, 11-22
- LIST command, 11-18
  - CARD file, 11-18
  - performed on a viewed database, 11-18
  - PRINTER option, 11-5, 11-18
  - REMOTE option, 11-5, 11-18
  - syntax, 11-18
- LOAD command, 11-19
  - CARD file, 11-19
  - EXCLUSIVE = TRUE option, 11-19
  - syntax, 11-19
- normal EOT, 11-6
- options, 11-23
  - LIMIT, 11-23
  - LIST, 11-23
  - WARNSUPR, 11-24
- REINITIALIZE command, 11-16
  - CARD file, 11-16
  - syntax, 11-16
- restarts, 11-6
  - conflicting executions, 11-6
  - restart information, 11-6
- updatable viewed databases, 11-3
- UPDATE command, 11-13
  - CARD file, 11-13, 11-15
  - copying and updating a database, 11-16
  - GENERATE option, 11-14
  - syntax, 11-13
  - updating software levels, 11-15
  - VALIDATE option, 11-15
- UPDATE/GENERATE run, 11-14
- UPDATE/VALIDATE run, 11-15
- utility files, 11-20
- validated databases, 11-2
- viewed databases, 11-3
  - updatable, 11-3
- single-valued DVA mapping, 9-22
  - default, 9-22
  - example, 9-26
  - options, 9-23
    - SIZE, 9-23, 9-24
    - STORED, 9-23, 9-25
  - storage compaction options, 9-23
  - syntax, 9-23
- <single-valued DVA option>, 9-23, B-9
- single-valued EVA mapping options, 9-39
  - syntax, 9-39
- <single-valued EVA option>, 9-39, B-9
- special cases, (*See* tradeoffs and special cases)
- SQL, 1-2
- string, 3-11
  - graphic format, 3-11, 3-12
  - hexadecimal format, 3-12
  - 16-bit character strings, 3-12
- string concatenation, implicit, 3-12
- <string subrange>, 4-11, B-4
- <string type>, 4-3, B-4
- Structured Query Language (SQL), 1-2
- subclass changes, 12-6
- <subclass declaration>, 5-6
- subclass declarations, (*See* subclasses)
- subclass mapping options, 9-47
  - DATASET-OPTIONS, 9-51
  - examples, 9-48, 9-50, 9-52
  - MAPPING, 9-47
    - DISJOINT-DATASET, 9-50
    - VARIABLE-FORMAT, 9-48
- <subclass options>, 9-47, B-9
- subclasses, 5-6
  - changes, 12-6
  - defining, 5-6
  - example, 5-8
  - syntax, 5-6

subrange constraints, 4-4  
 subranges  
     character, 4-6  
     integer, 4-5  
     real, 4-5  
 subrole DVA, 5-10  
 superclasses, 5-2  
 SURROGATE mapping option, 9-13  
     attribute scheme, 9-15  
     TIMESTAMP scheme, 9-14  
 surrogates, 5-1, 9-13  
 <symbolic subrange>, 4-17, B-5  
 syntax, ODL  
     <access constraints>, 8-10, B-7  
     <access declaration>, 8-2, B-7  
     <access structure>, 8-2, B-7  
     <all database>, 8-2, B-7  
     <attr options>, 9-20, B-8  
     <attributes>, 5-10, B-5  
     <base class declaration>, 5-4, B-5  
     <base class options>, 9-13, B-8  
     <character subrange>, 4-6, B-4  
     <class attributes>, 5-8, B-5  
     <compound type>, 4-3, B-4  
     <data set options>, 9-17, B-8  
     <data type>, 4-2, B-3  
     <data-valued attribute>, 5-11, B-5  
     <date subrange>, 4-8, B-4  
     <declarations>, 3-1, B-3  
     <dictionary-options declaration>, 10-1, B-10  
     <DMSII mapping-options declaration>, 9-2, B-8  
     <double subrange>, 4-9, B-4  
     <DVA options>, 5-11, B-5  
     <entity-valued attribute>, 5-17, B-6  
     <EVA options>, 5-18, B-6  
     <global options>, 9-3, B-8  
     <index declaration>, 6-2, B-6  
     <index options>, 9-53, B-9  
     <initial value>, 5-11, B-6  
     <integer subrange>, 4-5, B-3  
     <multivalued DVA option>, 9-29, B-9  
     <multivalued EVA option>, 9-40, B-9  
     <number format>, 4-8, B-4  
     <permission declaration>, 8-10, B-7  
     <real subrange>, 4-5, B-3  
     <set options>, 9-37, B-9  
     <single-valued DVA option>, 9-23, B-9  
     <single-valued EVA option>, 9-39, B-9  
     <string subrange>, 4-11, B-4  
     <string type>, 4-3, B-4  
     <subclass declaration>, 5-6  
     <subclass options>, 9-47, B-9

    <symbolic subrange>, 4-17, B-5  
     <time subrange>, 4-7, B-4  
     <user-defined subrange>, 4-17, B-5  
     <user-defined type declaration>, 4-18, B-3  
     <verify declaration>, 7-2, B-7  
 syntax, SIM Utility commands  
     <ADD command>, 11-7, B-1  
     <CHANGE command>, 11-9, B-1  
     <DB title>, 11-7, B-1  
     <LIST command>, 11-18, B-2  
     <LOAD command>, 11-19, B-2  
     <REINITIALIZE command>, 11-16, B-2  
     <UPDATE command>, 11-13, B-1  
     <utility commands>, 11-7, B-1

## T

target class, 5-16  
 time, 3-10  
 <time subrange>, 4-7, B-4  
 TIMESTAMP scheme, 9-14  
 tradeoffs and special cases, 2-29  
     compounds versus classes and EVAs, 2-29  
     conflicting views, 2-30  
     historic data, 2-30  
     multiple inheritance, 2-29  
     null attributes versus new subclasses, 2-29  
     transitional constraints, 2-29  
 trailing semicolons in a schema, 3-3  
 transitional constraints, 2-29  
 type chain, 12-2  
 types, 4-1  
     subrange constraints, 4-4  
     user-defined types, 2-20  
         examples, 4-19

## U

underscores and hyphens, 3-3, A-1  
 UNIQUE indexes, 2-21  
     restrictions, 6-4  
 unnecessary classes, 2-26  
 updatable viewed databases, 11-3  
 UPDATE command, 11-13  
     CARD file, 11-13, 11-15  
     copying and updating a database, 11-16  
     differences from CHANGE command, 11-15  
     GENERATE option, 11-14  
     syntax, 11-13  
     updating software levels, 11-15

- VALIDATE option, 11-15
- <UPDATE command>, 11-13, B-1
- UPDATE/GENERATE run, 11-14
- UPDATE/VALIDATE run, 11-15
- updating a database, 11-13
- <user-defined subrange>, 4-17, B-5
- user-defined type changes, 12-4
- <user-defined type declaration>, 4-18, B-3
- user-defined types
  - changes, 12-4
  - examples, 4-19
- utility commands, 11-7
  - ADD, 11-7
    - CARD file, 11-7
    - GENERATE option, 11-8
    - syntax, 11-7
  - CHANGE, 11-9
    - CARD file, 11-9
    - GENERATE option, 11-10
    - syntax, 11-9
    - VALIDATE option, 11-12
  - LIST, 11-18
    - CARD file, 11-18
    - performed on a viewed database, 11-18
    - PRINTER option, 11-5
    - REMOTE option, 11-5
    - syntax, 11-18
  - LOAD, 11-19
    - CARD file, 11-19
    - EXCLUSIVE = TRUE option, 11-19
    - syntax, 11-19
  - REINITIALIZE, 11-16
    - CARD file, 11-16
    - syntax, 11-16
  - UPDATE, 11-13
    - CARD file, 11-13
    - copying and updating a database, 11-16
    - GENERATE option, 11-14
    - syntax, 11-13
    - VALIDATE option, 11-15
- <utility commands>, 11-7, B-1
- utility files, 11-20
  - CARD, 11-20
  - DASDL, 11-20
  - DDLRESULTS, 11-21
  - LINE, 11-22
- utility options, 11-23
  - LIMIT, 11-23
  - syntax, 11-23

- LIST, 11-23
  - syntax, 11-23
- WARNSUPR, 11-24
  - syntax, 11-24
- Utility, SIM, (*See* SIM Utility)

## V

- VALIDATE option
  - for
    - ADD command, 11-9
    - CHANGE command, 11-12
    - UPDATE command, 11-15
- VALIDATE run, 11-2
- validated databases, 11-2
- verifies, 7-1
  - Boolean result, 7-4
  - changes, 12-13
  - condition, 7-4, 7-5
  - database schema change, 7-5
  - examples, 7-5
  - how used in a database, 7-4
  - locking implications, 7-4
  - OML global selection expression, 7-4
  - reinitializing a database that contains
    - verifies, 11-17
  - restriction on adding or changing, 7-5
  - syntax, 7-2
  - techniques for using, 7-1, 7-2
  - verify condition, 7-4, 7-5
  - when to declare, 7-1
  - when verifies are valid, 7-5
- verify changes, 12-13
- <verify declaration>, 7-2, B-7
- verify declarations, (*See* verifies)
- viewed databases, 11-3
  - updatable, 11-3

## W

- WARNSUPR option, 11-24
  - syntax, 11-24
- WFL (Work Flow Language), 1-4, 11-5
- Work Flow Language (WFL), 1-4, 11-5