

Technical Report

CMU/SEI-89-TR-16
ESD-TR-89-24

2

Carnegie Mellon University
Software Engineering Institute

AD-A215 846

Guidelines for the Use of the SAME

Marc H. Graham

May 1989

DTIC
ELECTE
DEC 12 1989
S B D

DISTRIBUTION STATEMENT A

Approved for public release;
Distribution Unlimited

89 12 11

0 0 5

Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admissions and employment on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders. In addition, Carnegie Mellon University does not discriminate in admissions and employment on the basis of religion, creed, ancestry, belief, age, veteran status or sexual orientation in violation of any federal, state, or local laws or executive orders. Inquiries concerning application of this policy should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6664 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Technical Report

CMU/SEI-89-TR-16

ESD-TR-89-24

May 1989

Guidelines for the Use of the SAME



Marc H. Graham

Ada SQL Project

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the


SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER


Charles J. Ryan, Major, USAF
SEI Joint Program Office

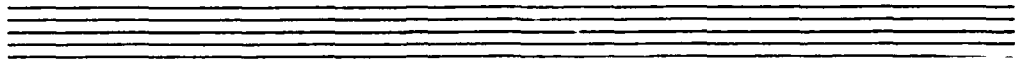
This work is sponsored by the U.S. Department of Defense.

Copyright © 1989 Carnegie Mellon University

This document is available through the Defense Technical Information Center. DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Service. For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161.

Use of any trademarks in this report is not intended in any way to infringe on the rights of the trademark holder.



SAME Software Order Form

Name _____

Organization _____

Mailing Address _____

City _____ State _____ Zip _____

Country _____ Phone () _____

E-Mail Address (if available) _____

- Distribution Medium:
- | | |
|--|---|
| <input type="checkbox"/> UNIX Tar Format | <input type="checkbox"/> MS-DOS Format |
| <input type="checkbox"/> TK50 Cartridge | <input type="checkbox"/> 5 1/4" Floppy Disk |
| <input type="checkbox"/> 1/4" Tape Cartridge | |
| <input type="checkbox"/> VMS Backup Format | |
| <input type="checkbox"/> TK50 Cartridge | |

Remit Amount: \$100.00 for US addresses, \$115.00 for foreign addresses.

Remit Procedure: All checks or purchase orders should be made payable to *Carnegie Mellon University Software Engineering Institute*. Please return this completed form along with your payment to:

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
USA
Attn: Business Services Division (Box SAME)

If you have questions, please contact the SEI Resource Center at (412) 268-5800.

Table of Contents

1. Introduction	1
1.1. Overview of the SAME Method	1
1.2. An Example of the SAME Method	8
1.3. Structure of This Document	15
2. The SAME Typing Model	17
2.1. Concrete Types	19
3. Developing the Abstract Domains	23
3.1. The SAME Treatment of SQL Null Values	23
3.1.1. The Minimalist Approach	24
3.1.2. The Full SQL Approach	25
3.1.3. A Compromise Approach for Comparison Operators	26
3.2. The Image and Value Functions	29
3.3. Range Constraints and the Generic Sub-Packages	30
3.4. Character Data	32
3.5. Decimal Fixed Point Arithmetic	37
3.5.1. Basic Support	38
3.5.2. SQL Support	41
3.5.3. Range Constraints for Decimal Types	42
3.6. Data Types Not in the SQL_Standard	45
3.6.1. Ada Enumeration Types	45
3.6.2. Date/Time Types	47
3.7. Packaging the Type Definitions	51
3.8. The Package SQL_Base_Types_Pkg	55
4. The SAME Operational Model	59
4.1. Constructing an Abstract Interface	59
4.1.1. A Note on Typing Parameters	60
4.1.2. A Note on Naming and Packaging	62
4.2. Constructing an Abstract Module	62
4.3. Database Exceptional Conditions	63
4.3.1. The Packages SQL_Communications_Pkg and SQL_Database_Error_Pkg	64
4.3.2. Handler for SQL_Database_Error	66
4.4. Note on the Overloading of INDICATOR Parameters	67
5. Notes on Writing Application Programs Using the SAME Method	69
5.1. Design Rules	69
5.2. Visibility and the Use of <i>use</i>	69
5.3. Using Non-ASCII Character Sets	70
5.4. Handling the Null_Value_Error Exception	71

5.5. Simulating Predefined Attributes	71
5.6. Doing Type Conversions	72
5.6.1. Ada Explicit Type Conversions	72
5.6.2. Using Conversion Functions	73
5.7. Using Three-Valued Logic	74
5.8. Commenting Procedure Calls	75
6. The SAME Method Summarized	77
7. Building a SAME Application Without a Module Compiler	81
8. Some Detailed Examples	85
9. Advanced DBMS Applications	113
9.1. Dynamic SQL	113
9.2. SQL and Ada Tasks	126
References	131
A SAME Quick Reference List	133
A.1 Example Domains	133
A.2 Functions Available to the Application	134
B Glossary of Terms	137
C SAME Standard Package Listings	143
C.1 Introduction	143
C.2 Copyright Notice	144
C.3 SQL_System Specification	144
C.4 SQL_Standard Specification	145
C.5 SQL_Communications_Pkg Specification	145
C.6 SQL_Communications_Pkg Body	145
C.7 SQL_Exceptions Specification	146
C.8 SQL_Boolean_Pkg Specification	146
C.9 SQL_Boolean_Pkg Body	147
C.10 SQL_Int_Pkg Specification	148
C.11 SQL_Int_Pkg Body	151
C.12 SQL_Smallint_Pkg Specification	156
C.13 SQL_Smallint_Pkg Body	158
C.14 SQL_Real_Pkg Specification	163
C.15 SQL_Real_Pkg Body	166
C.16 SQL_Double_Precision_Pkg Specification	170
C.17 SQL_Double_Precision_Pkg Body	172
C.18 SQL_Decimal_Pkg Specification	177
C.19 SQL_Decimal_Pkg Body	184

C.20 SQL_Decimal Assembler Support (VAX)	208
C.21 SQL_Decimal Assembler Support (IBM)	215
C.22 SQL_Char_Pkg Specification	226
C.23 SQL_Char_Pkg Body	229
C.24 Subunit To_String	235
C.25 Subunit To_SQL_Char_Not_Null	235
C.26 SQL_Enumeration_Pkg Specification	235
C.27 SQL_Enumeration_Pkg Body	237
C.28 SQL_Database_Error_Pkg Specification	241
C.29 SQL_Database_Error_Pkg Body	24 i
C.30 SQL_Date_Pkg Specification	241
C.31 INGRES_Date_Pkg Specification	245

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

List of Figures

Figure 1-1:	Classical Approach to Database Access	2
Figure 1-2:	Modular Approach to Database Access	3
Figure 1-3:	The Manual Method	6
Figure 1-4:	The Automated Method	7
Figure 1-5:	An E-R Diagram for Parts and Suppliers	9
Figure 1-6:	The Parts-Suppliers Schema	9
Figure 1-7:	Some of the Abstract Domains as Ada Types	10
Figure 1-8:	Example Abstract Interface	11
Figure 1-9:	An Application Program Using an Abstract Interface	12
Figure 1-10:	Application Using Concrete Interface	13
Figure 1-11:	The Concrete Module for the Example	14
Figure 1-12:	Ada Specification of Concrete Module -- The Concrete Interface	14
Figure 1-13:	Body of the Abstract Interface -- The Abstract Module	16
Figure 2-1:	The SAME Typing Model	19
Figure 2-2:	The Package SQL_STANDARD	20
Figure 2-3:	The Package SQL_System	21
Figure 3-1:	Three-Valued Logic	26
Figure 3-2:	The Generic Subpackage Sql_Int_Ops	31
Figure 3-3:	The Generic Subpackage SQL_Char_Ops	34
Figure 3-4:	The Generic Subpackage SQL_Decimal_Ops	44
Figure 3-5:	The Package Specification SQL_Enumeration_Pkg	48
Figure 3-6:	The Domain Packages for Suppliers-Parts	54
Figure 3-7:	The Domain Packages for Suppliers-Parts, cont'd.	55
Figure 3-8:	The Package SQL_Base_Types_Pkg	57
Figure 4-1:	Parameter Kinds (with Modes)	60
Figure 4-2:	The Abstract Module Procedure Calculate_Weight	63
Figure 4-3:	Package Specifications for Sql_Communications_Pkg and SQL_Database_Error_Pkg	65
Figure 6-1:	SAME Application Package Structure	79
Figure 7-1:	Concrete_Mod for Alsys	82
Figure 7-2:	Concrete_Mod for Verdix	82
Figure 8-1:	A Block Diagram of the Example	86
Figure 8-2:	The SQL Procedure for Example_A	88
Figure 8-3:	The Abstract Module for Example_A	88
Figure 8-4:	Example_A (Part I)	89
Figure 8-5:	Example_A (Part II)	90
Figure 8-6:	The Abstract Module Body for Example_A	91
Figure 8-7:	The Conversions Package	93

Preface

Overview of Document and Intended Audience

These guidelines describe the Structured Query Language (SQL) Ada Module Extensions, a method for the construction of Ada applications that access database management systems whose data manipulation language is SQL. The SAME is not a tool set, it is a method of program design and development. There is a set of support software, called the SAME standard packages, which are needed by applications using the SAME.

As its name implies, the SAME extends the capabilities of the Module language defined in the ANSI SQL standard to fit the needs of Ada. The defining characteristic of the use of the module language is that the SQL statements appear together, physically separated from the Ada application, in an object called the *module*. The Ada application accesses the module through procedure calls.

The primary audience for this document consists of application developers and technicians creating Ada applications for SQL database management systems. The document contains a complete description of the SAME, including its motivation. It is not intended as a programmer's guide. Organizations using the SAME may wish to create such a guide from this document.

The reader of this document is expected to be familiar with both Ada and SQL, at some level of detail. An attempt has been made to make the document accessible to readers who are not experts in either language. Technical details are explained under the assumption that the reader has a general understanding of both languages.

A Note on the Code in This Document

All of the Ada code in this document has been compiled, in many cases on more than one compiler, and the great bulk of it has been tested. Exceptions to this rule are noted in the text. The code in Appendix C has been exhaustively tested. The SQL code in the document has also been tested, but not in the exact form shown. However, the processes of transcribing the code into the document and editing it for improved readability may have inadvertently introduced errors. The code in the appendix was copied into the document without modification and should thus be less likely to contain errors.



Acknowledgments

This document would never have been created were it not for the efforts of the Structured Query Language (SQL) Ada Module Extensions Design Committee (SAME-DC). This volunteer committee of users, database and compiler vendors, and recognized experts has been meeting regularly since May 1988. The hard work and heated discussions of those meetings effectively shaped this document.

The following is a list of those people who attended SAME-DC meetings. Companies are listed for information purposes only. In no case should the opinions in this document be considered those of the companies listed, nor of any individual in this list.

Name	Organization
Judith Bamberger	Software Engineering Institute
Wanda B. Barber	USA - ISS - Development Center Lee
Stowe Boyd	Meridian
Bill Brykczynski	Institute for Defense Analyses
Scott L. Burns	Computer Science Corp
Janet E. Edwards	Headquarters, USA Information Service Support Center
Robert Firth	Software Engineering Institute
Neil Goodman	RTI
Marc Graham	Software Engineering Institute
Nabil Hijazi	MITRE Corp
Jeff Ives	Compass
Phillip R. Joiner	USA - ISS - Development Center Lee
Arthur Keller	Stanford University
Gary M. Lichvar	U.S. Army - ISS - Development Center Lee
James Metcalfe	Hewlett-Packard
Jim Moore	IBM
Dit Morse	Oracle Corp
Susan Philips	Lockheed Software Technology Center
Judith Richardson	US Army Communications Electronics Command
Paul Sciabica	Cullinet Software
Phil Shaw	IBM
John Steensen	Applied Data Research, Inc.
S. Tucker Taft	Intermetrics
Pat Timpanaro	Compass
Keith Usher	IBM
Eugene Vasilescu	Grumman Data Systems
Hector Villarreal	Sybase Corporation
Kurt Wallnau	UNISYS
Tom Wheeler	USA Communications Electronics Command
Bill Wood	Software Engineering Institute
Dale Worley	Compass
Greg Zelesnik	Software Engineering Institute

The author would particularly like to thank Stowe Boyd for his help in publicizing this work, and Arthur Keller, Susan Philips, and Tucker Taft for hosting meetings of the SAME-DC. Special thanks to Greg Zelesnik, who is responsible for much of the code in this document and much of the work in verifying the code's correctness.

This work was financially supported by the Ada Joint Program Office (AJPO). The author and the SAME Design Committee wishes to thank Ginny Castor, David Taylor, and Glenn Hughes for their support.

Guidelines For the Use of the SAME

Abstract. These guidelines describe the Structured Query Language (SQL) Ada Module Extensions, or SAME, a method for the construction of Ada applications that access database management systems whose data manipulation language is SQL. As its name implies, the SAME extends the module language defined in the ANSI SQL standard to fit the needs of Ada. The defining characteristic of the use of the module language is that the SQL statements appear together, physically separated from the Ada application, in an object called the module. The Ada application accesses the module through procedure calls.

The primary audience for this document consists of application developers and technicians creating Ada applications for SQL database management systems. The document contains a complete description of the SAME, including its motivation.

1. Introduction

The SQL Ada Module Extensions (SAME) method of constructing database application programs in Ada is based on the SQL module language [2]. The method extends the features of the module language by exploiting the capabilities of Ada. This results in robust application programs written in a style suitable to Ada. The SAME treats SQL in much the same way that Ada treats other foreign languages; that is, it imports complete modules, not language fragments.

1.1. Overview of the SAME Method

In the classical approach to database access from application programming languages [3], the programmer prepares a single text containing statements from two different languages: the programming language and a database language. These two subtexts are disentangled by a so-called *preprocessor*, which outputs the programming language text in which the database statements have been replaced with procedure calls. This text can be processed by the programming language compiler. A diagram of this process is given in Figure 1-1.

A programmer using a modular method such as the SAME does not prepare such a mixed text. Instead, he prepares a compilable Ada program in which database services are accessed via procedure calls. The bodies of those procedures are defined by SQL statements collected into a separate text called a *module*. The process is diagrammed in Figure 1-2.

As Ada database application programs written with the SAME are written in pure Ada, there is no need for an Ada/SQL preprocessor. Ada-sensitive editors and debuggers can be used to create these applications. Since the database interactions are written in standard SQL, they can be processed by existing SQL tools. There is no need for programmers to learn new syntax and semantics; no new system software need be written, maintained, and

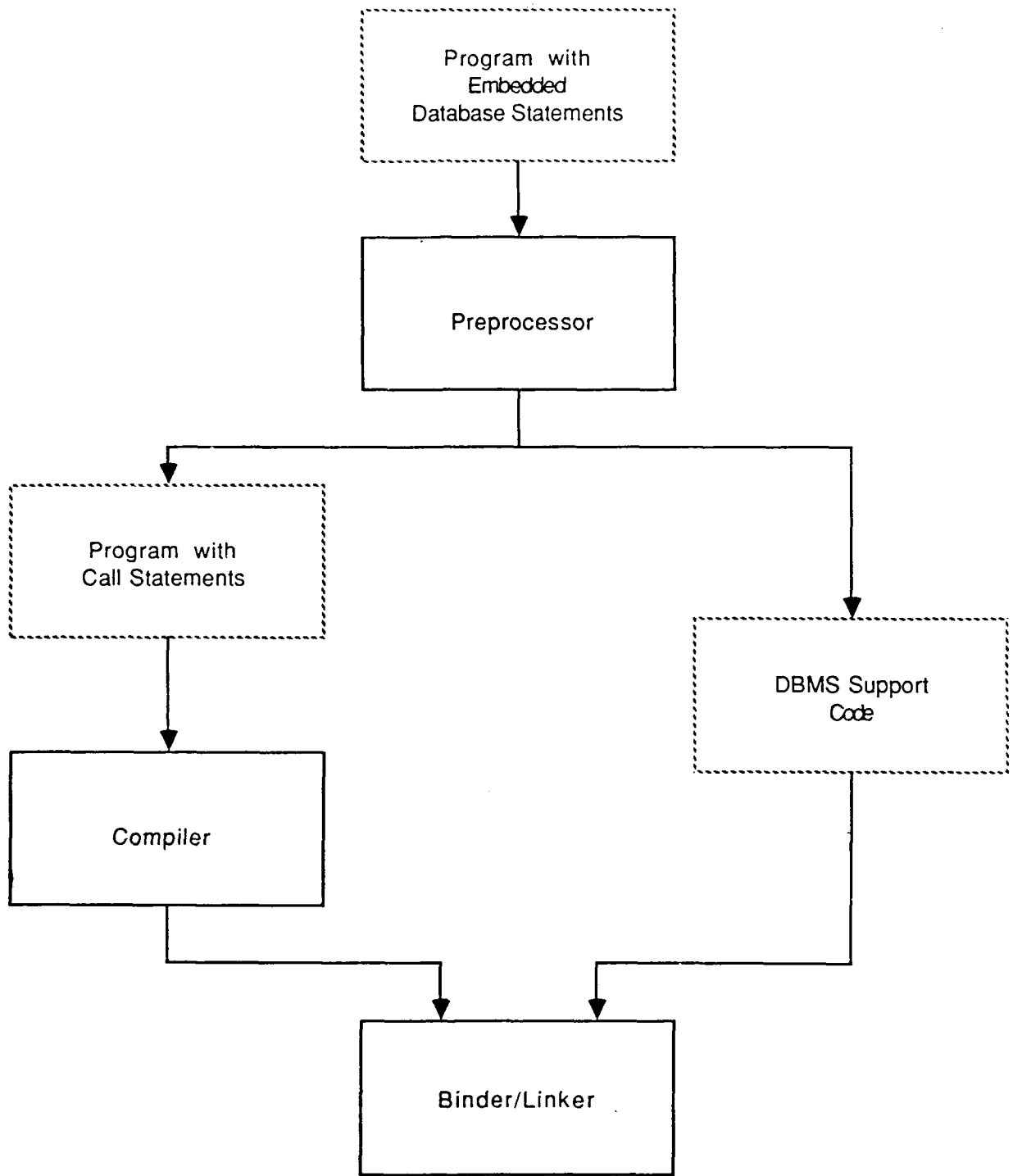


Figure 1-1: Classical Approach to Database Access

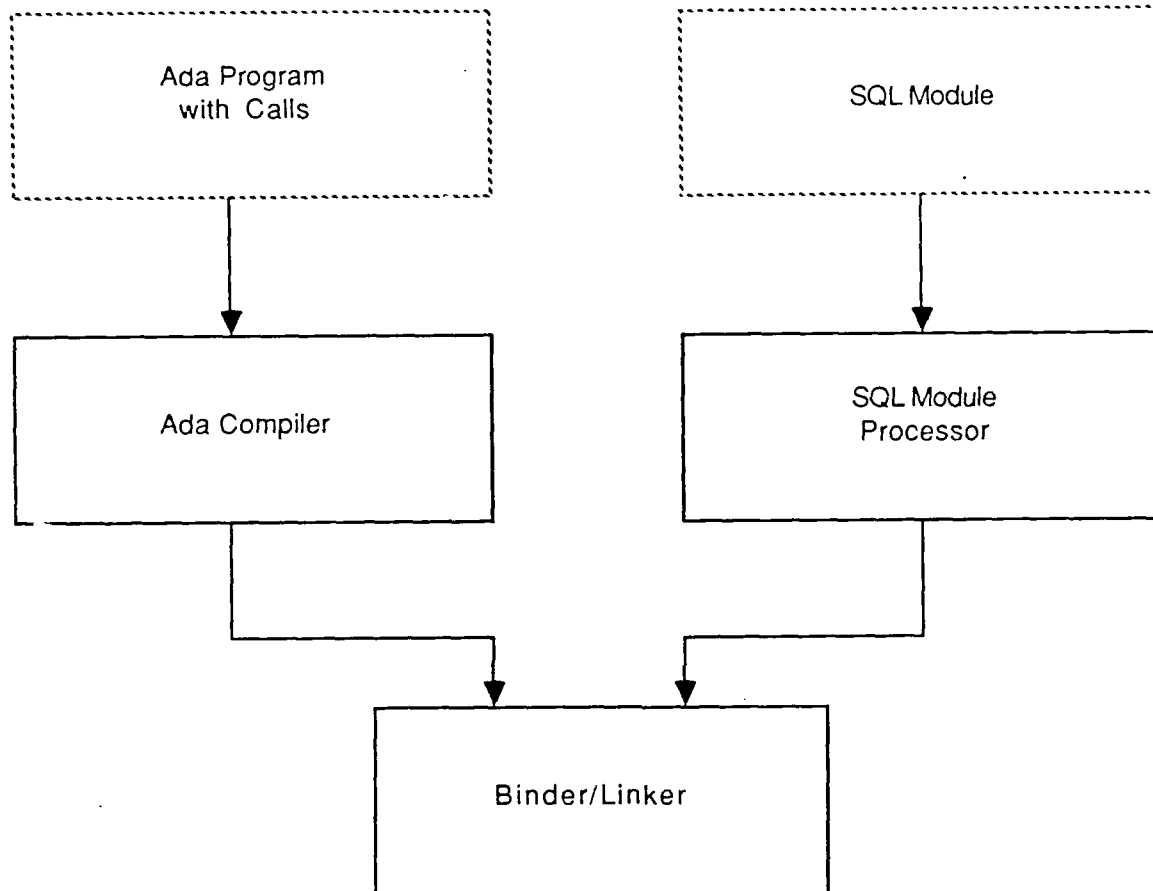


Figure 1-2: Modular Approach to Database Access

ported to process a new syntax and semantics for SQL.¹ In this regard, the SAME treats Ada and SQL as equals. The SAME interfaces two existing standards and their implementing software. It does not attempt to create an "ideal" Ada DBMS. Rather, it allows access to existing, commercial DBMS in a manner which exploits the tools and capabilities of the DBMS.

Using the preprocessor approach to database application programming as shown in Figure 1-1, the application programmer must know the syntax and semantics of not only the programming language but also the database language. These are rarely identical or even similar; certainly not in the case of Ada and SQL. The programmer must think in two different ways as he alternates between Ada and SQL. In such non-modular approaches, the application programmer must understand not only the logic of the application, but also the logical design of the stored database. He must know not only what information services the application program requires of the database, but also how the database can be made to provide those services.

Modular approaches, such as the SAME, make it possible for the application and database programming tasks to be assigned to different programmers. For development organizations which are large enough to afford this specialization of roles, there are benefits in reduced training costs and greater productivity. In the case that the same programmer creates the Ada application and the SQL module, he is able to separate the concerns of the application logic and the database logic. When designing or writing the application he can ignore the issues of database interaction; when dealing with the database he can concentrate solely on it. In both cases, since the resulting Ada application program contains no SQL, it is isolated from changes in the database structure and the SQL statements. This isolation decreases the cost of maintenance and porting.

Large, complex database applications have extensive design phases. Modular approaches such as the SAME are particularly well suited for such applications. The module makes the database services needed by the application visible. It is an application-specific, DBMS-independent interface between the database and the application, which is naturally treated during the design as a design object. The dependence of the application on the database can be controlled more easily since it is more visible, not scattered throughout the application as in non-modular approaches. The module is an *external schema* [6], a "simple user view, tailored to the requirements of a specific application" [8].

The benefits of modular interfaces are summarized in the following list.

- Maintenance and porting costs are reduced by the isolation and separation of the Ada code from the SQL code. The application - database interaction is elevated to the status of a design object. This makes it easier to manage and control.

¹The method proposed by the Institute for Defense Analysis (IDA) [12] does not embed SQL into Ada in the standard sense, but it does produce application programs containing intermixed application and database logic. This is done by modifying the syntax and semantics of SQL so that it appears as compilable Ada code. The necessary support packages and system software are expensive in development, compilation, and runtime costs, although accurate figures are not available. By separating the Ada and the SQL and allowing each to be processed by pre-existing processors, the SAME avoids these modifications and expenses entirely.

- The potential exists for increased specialization of the software development team. Fewer programmers need to know the details of the database design. This can lead to improvements in team productivity.
- Ada application programs are written in compilable Ada, preserving the use of syntax-directed editors, etc. There is no need for pre-processing. There is no need to develop any new syntax nor system software; these methods can be used with existing tools.²

The SAME is a specialization of the modular approach particular to the needs of Ada. The benefits which it brings to database applications written in Ada are:

- **The Ada typing model.** Using the SAME method, the Ada program views the database through the abstract type facilities of Ada. Type derivation and sub-typing are available as are range constraints to control runtime behavior and inappropriate operand usage.
- **A safe treatment of null values.** SQL supports partial and incomplete information through the use of the null value. The null value is a concept foreign to Ada, as it is to most programming languages. Through the use of Ada's data abstraction mechanism, the SAME brings a measure of incomplete information processing to Ada while ensuring that null values are never used as though they were not null.
- **A simple, robust, yet flexible treatment of database exceptional conditions.** SQL database management systems signal the occurrence of exceptional events, such as hardware failure, through a status code field. The meanings of the values of that field are not set by the standard; each implementation presents a different set of values. Usually the application program cannot recover from any of these conditions. The SAME treatment of exceptional conditions presents a failure-free DBMS to the application program; if an SQL statement encounters an unexpected condition, an exception is raised and an appropriate error message is generated. This simplifies the application programmer's job and ensures uniform treatment of errors. On the other hand, the SAME allows applications which need to do some or all of their own error processing full access to the DBMS facilities.

The features in the above list are implemented in a thin interface layer, called the *abstract module*. In Figure 1-3, the concrete module is the object containing solely SQL statements as might be processed by an SQL module language compiler.³ The abstract module serves to transform data and procedural abstractions of the Abstract and Concrete Interfaces of Figure 1-3. The architecture of Figure 1-3 is specific to the manual implementation of the SAME method. The SAME Design Committee (SAME-DC) is engaged in the task of specifying the syntax and semantics of a tool to assist in the construction of abstract interfaces. When such tools become available, the situation simplifies to that given in Figure 1-4. The SAME method is valuable without such tooling, but is easier to use with it.

²This will depend on the tool sets supplied by particular Ada compiler and DBMS vendors. It is always possible to use the method; these tool sets may make it easier.

³As of this writing, there are no compilers for the SQL module language, although there are some under development that are due to be released soon. In later chapters we show how to build applications in SAME without a module language compiler.

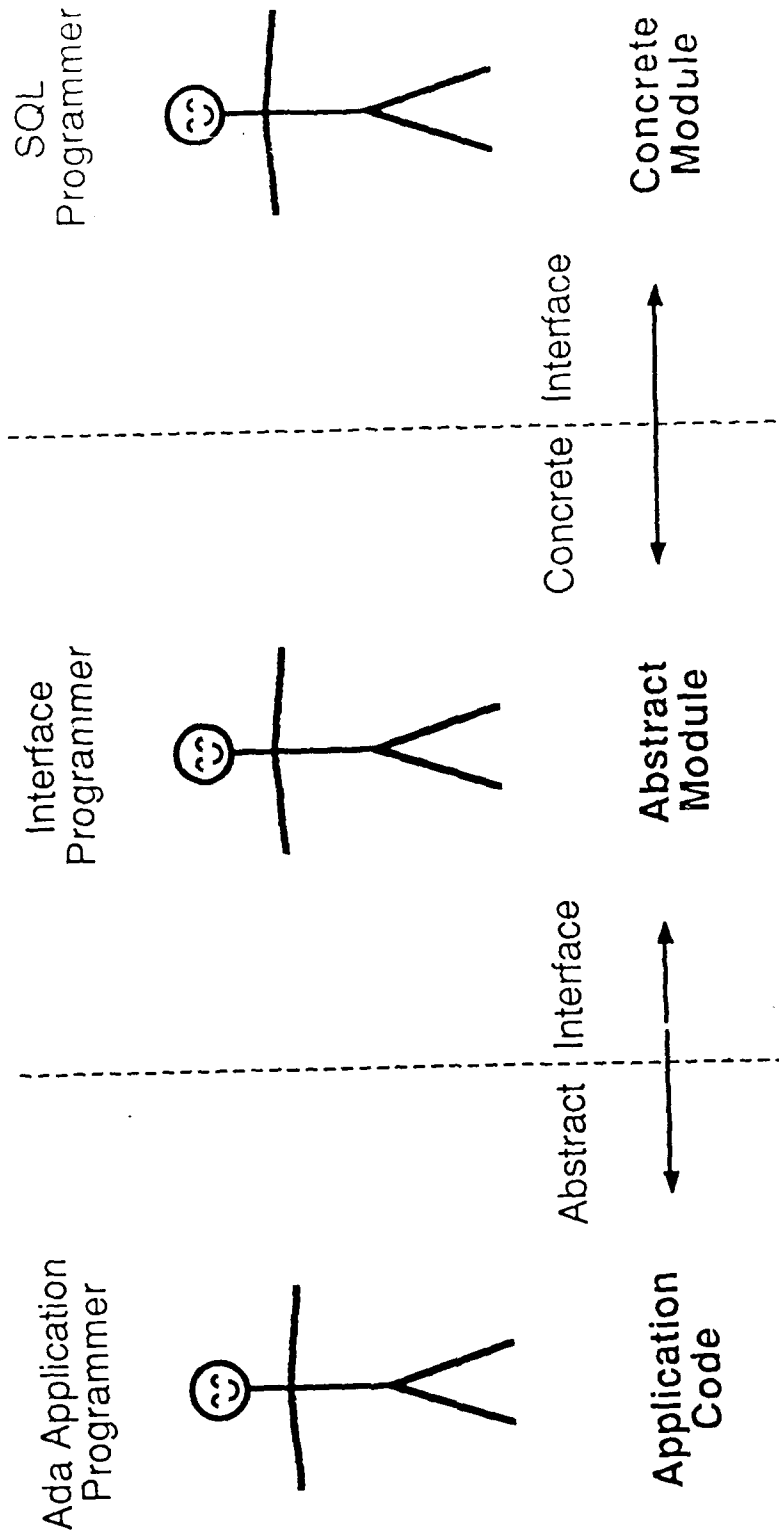


Figure 1-3: The Manual Method

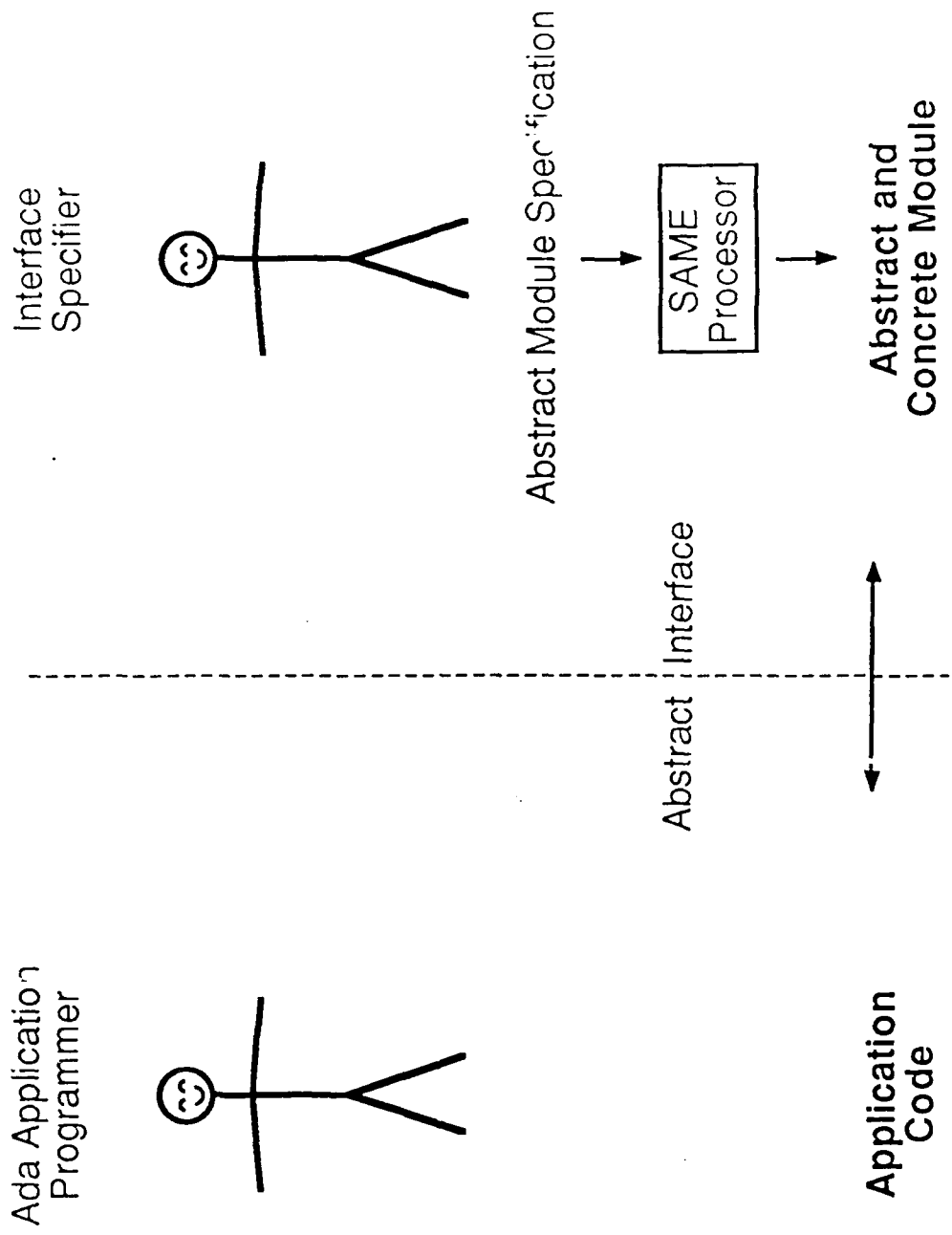


Figure 1-4: The Automated Method

1.2. An Example of the SAME Method

This section provides, by way of example, an overview of the SAME in use. It is meant to provide the reader with an intuitive feel for the method. Later sections provide the details.

Use of the SAME begins during the process of database design.⁴ Early in that process the designer delineates the *abstract domains* of his database. The notion of an abstract domain is very similar to the notion of an abstract type. However, the Ada definition of an abstract domain requires more than a single Ada type definition, as will be shown. Hence, a new term was needed to define this concept.

Abstract domains are objects in the real world that are reflected in the information system which models that world. They are also the objects from which the database structures, that is, the relations, will be built. They describe, *inter alia*, the value sets which may appear in database columns. Like Ada types, abstract domains serve to distinguish differing denotations of a concrete value; the value "1" as an employee number is not the same as the value "1" as a department number, for example.

Abstract domains tend to lose their identities in the SQL schema due to SQL's weak typing model. Ada's typing model allows these domains to retain their identities and the SAME exploits that power.

Entity relationship diagrams [7] are a popular database design aid. Figure 1-5 contains such a diagram, describing the parts-suppliers database of C. J. Date [9]. The diagram describes two entities: Suppliers, uniquely identified by Number and having attributes Name, Status and City; Parts, also uniquely identified by Number, with attributes Name, Color, Weight, and City. (The city of a supplier is the city in which the supplier is located; the city of a part is the city in which the part is stored.) The diagram also recognizes one relationship, Order, which relates a supplier and a part, and has the attribute Quantity.

Designing the abstract domains in a database design is much like designing the abstract data types of an Ada program. A good rule of thumb to follow is the *comparison rule*. If it makes sense to compare values of two different Ada variables or database attributes, then they probably have the same Ada type or abstract domain. For example, it makes no sense to compare supplier numbers to part numbers; part number one is utterly different from supplier number one. The same is true for supplier and part names. On the other hand, supplier cities and part cities have the same abstract domain; "Pittsburgh" is "Pittsburgh" whether a supplier or a part is located there. Thus, the abstract domains in Figure 1-5 are supplier number (SNO), supplier name (SNAME), STATUS, CITY, part number (PNO), part name (PNAME), COLOR, WEIGHT, and quantity (QTY).

The SQL schema for this database is given in Figure 1-6. Notice that the abstract domains have been obscured.⁵ SNO and PNO have the same data type, although they take values from distinct abstract domains. The SAME Ada types for these columns makes the distinc-

⁴These steps are easily retro-fitted to a pre-existing database design.

⁵In this case, the domains have been preserved in the attribute names. In general, relational database design methods, and SQL in particular, do not recognize abstract domains.

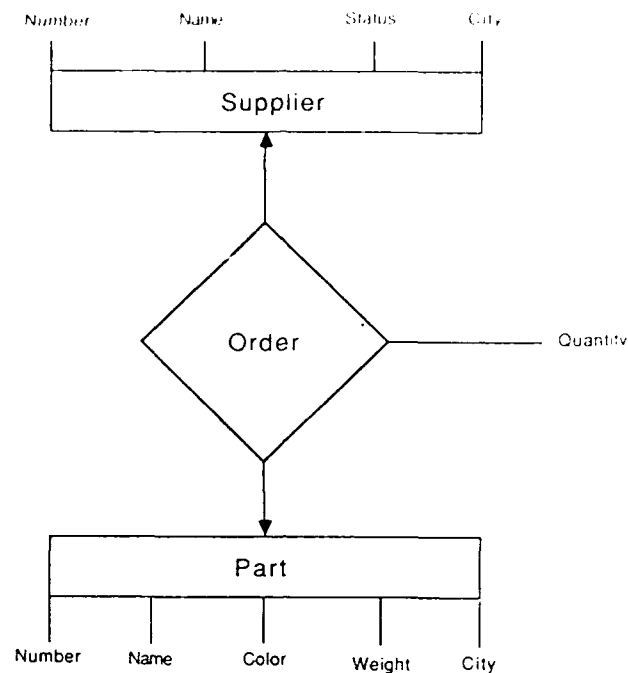


Figure 1-5: An E-R Diagram for Parts and Suppliers

```

CREATE TABLE S ( SNO CHAR(5) NOT NULL,
                  SNAME CHAR(20),
                  STATUS INT,
                  CITY CHAR(15),
                  UNIQUE ( SNO ) )
  
```

```

CREATE TABLE P ( PNO CHAR(5) NOT NULL,
                  PNAME CHAR(20),
                  COLOR CHAR(6),
                  WEIGHT INT,
                  CITY CHAR(15),
                  UNIQUE ( PNO ) )
  
```

```

CREATE TABLE SP ( SNO CHAR(5) NOT NULL,
                  PNO CHAR(5) NOT NULL,
                  QTY INT,
                  UNIQUE ( SNO, PNO ) )
  
```

Figure 1-6: The Parts-Suppliers Schema

tion apparent. The full set of type declarations for some of these abstract domains is given in Figure 1-7. The meanings of these definitions is not immediately obvious; they will be explained in Chapters 2 and 3.

Although the SAME is a method for interfacing Ada and SQL and not a tool set, it does have underlying support software. This software is known collectively as the SAME standard packages. The packages SQL_Char_Pkg and SQL_Int_Pkg are two of these packages. A

complete listing of the specification and bodies of these packages, along with a quick reference guide to them, are attached as Appendix A.⁶

```
with SQL_Char_Pkg; with SQL_Int_Pkg;
package Example_Definitions is

  type PNONN_Base is new SQL_Char_Pkg.SQL_Char_Not_Null;
  subtype PNO_Not_Null is PNONN_Base (1..5);
  type PNO_Base is new SQL_Char_Pkg.SQL_Char;
  subtype PNO_Type is PNO_Base (PNO_Not_Null'Length);
  package PNO_Ops is new
    SQL_Char_Pkg.SQL_Char_Ops(PNO_Base, PNONN_Base);

  type CITYNN_Base is new SQL_Char_Pkg.SQL_Char_Not_Null;
  subtype CITY_Not_Null is CITYNN_Base (1..15);
  type CITY_Base is new SQL_Char_Pkg.SQL_Char;
  subtype CITY_Type is CITY_Base (CITY_Not_Null'Length);
  package CITY_Ops is new
    SQL_Char_Pkg.SQL_Char_Ops(CITY_Base, CITYNN_Base);

  type Status_Not_Null is new SQL_Int_Pkg.SQL_Int_Not_Null;
  type Status_Type is new SQL_Int_Pkg.SQL_Int;
  package Status_Ops is new
    SQL_Int_Pkg.SQL_Int_Ops(Status_Type, Status_Not_Null);
end Example_Definitions;
```

Figure 1-7: Some of the Abstract Domains as Ada Types

In Figure 1-7, each of the illustrated abstract domains has two Ada types. One of the types, with the suffix `_Not_Null`, is a visible Ada type; thus `Status_Not_Null` is an integer type; `PNO_Not_Null` is a one dimensional array, of a character type.⁷ The other type, with the suffix `_Type`,⁸ is a limited private type. This type provides an encapsulation of the SQL null value. A full range of comparison and, for numeric types, arithmetic operators are defined for these types. These operators implement the semantics of the corresponding SQL operator, which is defined for the null value. The majority of these operators are derived, using Ada derivation, from those defined in the SAME standard packages. The few operators which cannot be derived in this way are generated by the generic packages illustrated in Figure 1-7. This is done to reduce compilation time and runtime storage requirements.

In the remainder of these guidelines, the two types which together with the package instantiation make up the declaration of an abstract domain are called the *visible Ada* or *_Not_Null* type and the *limited* or *_Type* type.

Once the database schema has been defined in Ada, subsequent steps of the SAME are application specific. Consider the following application: "For each part ordered from any supplier, print the part number and the names of cities in which some supplier with a status of *X* or greater is located. *X* is a runtime parameter." In order to implement this application, an Ada program will need three database procedures:

⁶The SEI will, for a limited time, distribute this software in machine-readable form. An order form is attached to this document.

⁷This type may be other than `Standard.Character`, as the database may store non-ASCII character strings.

⁸Section 3.4 explains the need for the structure of the character string type definitions.

1. An "open cursor" procedure which accepts the runtime parameter.
2. A "fetch" procedure to return the rows of part numbers and cities.
3. A "close cursor" procedure to be called when the application has exhausted all selected rows.

The program will also need a definition of the rows being passed to it. These procedure and row record definitions make up the abstract interface, the specification of the abstract module. That specification, for this example, is given in Figure 1-8.

```

with Example_Definitions; use Example_Definitions;
package Example_Interface is

  type Part_Nbr_City_Pairs is record
    Pno : PNO_Not_Null;
    City : City_Type;
  end record;

  -- All of these procedures may raise SQL_Database_Error

  procedure Open (Lower_Bound : Status_Not_Null);
    -- creates the relation of Part numbers and Cities
    -- where there exists some supplier, with status
    -- at least Lower_Bound, of that part in that city

  procedure Fetch (Tuple : in out Part_Nbr_City_Pairs;
    Found : out Boolean);
    -- returns the records of the relation created by open
    -- Found becomes False at end of table

  procedure Close;
    -- clean up procedure
end Example_Interface;

```

Figure 1-8: Example Abstract Interface

Once the abstract interface has been determined, the application program can be written. Figure 1-9 contains the application program. For that figure, assume that Status_IO is an instantiation of Integer_IO for the integer type Status_Not_Null. The functions Not_Null and to_unpadded_string are supplied by the SAME standard packages.

It is instructive to notice the differences between application programs using an abstract interface, as exemplified by Figure 1-9, and one using the concrete interface provided by the ANSI module language, as is shown in Figure 1-10. (In Figure 1-10, Example_Module is the Ada package name assigned to the concrete module, which is illustrated in Figure 1-11; SQL_Standard is a package defined in a revised version of the ANSI standard. See [16] [5] and Section 2-1. SQL_Int_IO is Integer_IO instantiated for SQL_Standard.Int.)

```

with Text_IO; use Text_IO;
with Status_IO; -- Integer_IO instantiated for Status_Not_Null
with Example_Interface; use Example_Interface;
with Example_Definitions; use Example_Definitions;
procedure Example is

  Status_Buffer : Status_Not_Null;
  Data_Record   : Part_No_City_Pairs;
  Record_Found  : Boolean;

begin
  put("Enter Status=> ");
  Status_IO.Get(Status_Buffer); new_line;
  put("Part Numbers and Cities for Status ");
  Status_IO.put(Status_Buffer); new_line;
  Open(Lower_Bound => Status_Buffer); -- create result table
  loop
    fetch(Data_Record, Record_Found); -- next record into buffer
    exit when not Record_Found; -- if exit taken, all done
    if Not_Null(Data_Record.City) then -- filter out unknown cities
      put_line(to_unpadded_string(Data_Record.Pno) & " " &
              to_unpadded_string(Data_Record.City));
    end if;
  end loop;
  close;
end Example;

```

Figure 1-9: An Application Program Using an Abstract Interface

These differences are summarized in the following list.

- Using an abstract interface, an application program treats rows of a table as an object of a record type. At the concrete interface, the components of a row are treated as individual parameters.
- Using an abstract interface, an application program sees the database through the abstract domains identified during database design. At the concrete interface, only the limited set of SQL types are present.
- Using an abstract interface, an application programmer may safely remain unaware of the SQL conventions for null values. At the concrete interface, separate indicator variables signal nullness. Obscure errors can result from mishandling these indicators. These errors cannot arise in programs using the SAME.
- Using an abstract interface, an application program does not see the SQLCODE parameter. This is the variable which holds the status code returned from every SQL statement execution. At the concrete interface, the application must check this parameter, understand it, and execute application supplied error processing if things go wrong. Obscure errors can result from not handling these DBMS exceptional conditions correctly. These errors are eliminated from programs using the SAME.

It is also worth noting that the abstract interface provides facilities which permit application programs to be indifferent to the encoding of the character data in the database. The concrete interface supports the use of non-ASCII characters but provides no mechanism for inter-converting them with ASCII characters. For example, the Ada explicit type conversions (that appear as arguments to the put_line call in Figure 1-10) assume that the DBMS stores ASCII character strings. In contrast, the corresponding portion of Figure 1-9 uses an abstract interface function (to_unpadded_string) which will convert the DBMS character set to

ASCII if needed. (The decision is made as part of the installation of the SAME support packages. See [14].)

```
with Text_IO: use Text_IO;
with SQL_Int_IO;
with Example_Module; use Example_Module;
with SQL_Standard;
procedure Example_at_Concrete_Interface is

  Status_Buffer : SQL_Standard.Int;
  Part_Number: SQL_Standard.Char(1..5);
  City: SQL_Standard.Char(1..15);
  SQLCODE : SQL_Standard.SQLCODE_Type;
  City_Indicator : SQL_Standard.Indicator_Type;

begin
  put("Enter Status=> ");
  SQL_Int_IO.Get(Status_Buffer); new_line;
  put("Part Numbers and Cities for Status ");
  SQL_Int_IO.put(Status_Buffer); new_line;
  Open(Status_Buffer, SQLCODE);
  if SQLCODE in SQL_Standard.SQL_Error then
    <application supplied error processing>
  else
    loop
      fetch(Part_Number, City, City_Indicator, SQLCODE);
      if SQLCODE = 0 then
        if City_Indicator >= 0 then
          put_line(string(Part_Number) & " " &
            string(City));
        end if;
      elsif SQLCODE in SQL_Standard.SQL_Error then
        <application supplied error processing>
        exit;
      elsif SQLCODE in SQL_Standard.Not_Found then
        exit;
      end if;
    end loop;
  end if;
  close;
end Example_at_Concrete_Interface;
```

Figure 1-10: Application Using Concrete Interface

There remains now only the task of creating the body of the abstract interface, also called the abstract module. The purpose of the procedures in that module is to form the bridge between the concrete interface and the abstract interface. It is assumed in this section that the concrete interface is supplied by a module language compiler that is compliant with the ANSI standard. The SAME does not depend on the existence of such compilers. Chapter 7 demonstrates the use of the SAME in environments without such compilers.

Figure 1-11 contains the specification of the concrete module for the example as it would be written in the module language. The Ada package specification corresponding to that module, according to the revised ANSI standard [5] [16], appears in Figure 1-12. The body of that package is implementation dependent; in particular, its form will depend on the tool set available for the DBMS is use. Finally, the abstract module, implementing the abstract interface on top of the concrete interface, appears in Figure 1-13.

```

Module Example_Module
Language Ada
Authorization Public

Declare X Cursor
For
  Select SP.PNO, S.City
  From SP, S
  Where SP.SNO = S.SNO
  And S.Status >= Input_Status;

Procedure X_Open
  Input_Status Int
  SQLCODE;
  Open X;

Procedure X_Fetch
  Part_Number Char(5)
  City Char(15)
  City_Indic Smallint
  SQLCODE;
  Fetch X into Part_Number, City INDICATOR City_Indic;

Procedure X_Close
  SQLCODE;
  Close X;

```

Figure 1-11: The Concrete Module for the Example

```

with SQL_Standard;
package Example_Module is

  procedure X_Open (Input_Status : SQL_Standard.Int;
                   SQLCODE : out SQL_Standard.SQLCODE_Type);

  procedure X_Fetch (Part_Number : out SQL_Standard.Char;
                   City : out SQL_Standard.Char;
                   City_Indic : out SQL_Standard.Indicator_Type;
                   SQLCODE : out SQL_Standard.SQLCODE_Type);

  procedure X_Close (SQLCODE : out SQL_Standard.SQLCODE_Type);

end Example_Module;

```

Figure 1-12: Ada Specification of Concrete Module -- The Concrete Interface

The detail in Figure 1-13 (for example, the purpose of the packages SQL_Communications_Pkg and SQL_Database_Error_Pkg) is explained in Chapter 4, which explains the construction of abstract modules. The outline of an abstract interface procedure body can be recognized in Figure 1-13. That outline is described by the following list.

1. The corresponding procedure in the concrete interface is called. Any parameters to that procedure are conveyed to the appropriate type in package SQL_Standard.
2. The resulting status code parameter (SQLCODE) is examined. If the value of that parameter lies in a set of expected values, control is returned to the application program. Otherwise, a standardized error processing routine is called and an exception is raised.
3. Values which may be null are checked for nullness, converted to the appropriate types for the application program and assigned to the output row record. Values which may not be null are placed directly into the output row record by the concrete procedure. (In the case of INSERT or UPDATE SQL statements, for which data flows from the application to the database, this set of steps occurs first.)

The fact that every abstract interface procedure body has a predictable structure makes them prime candidates for automatic generation. The SAME Design Committee hopes to create, in the near term, a notation enhancing the standard ANSI module language, within which abstract interfaces can be described and from which they can be generated. This is the idea behind Figure 1-4.

1.3. Structure of This Document

The remainder of these guidelines presents the SAME in detail. Chapters 2 and 3 tell the database designer how to describe the database in terms of the abstract types used by the SAME. Chapter 4 gives the information needed by the builder of abstract interface modules. Chapter 5 contains hints and suggestions for designers and programmers of applications using the SAME. Much of the information in Chapter 5 also appears elsewhere in the guidelines. It is repeated in Chapter 5 for the convenience of application programmers. Chapter 6 contains a condensed overview of the SAME. The bulk of this document assumes the existence of a compiler for the ANSI standard module language. Use of the SAME does not require such a compiler. Chapter 7 describes how the SAME can be used without a module language compiler. Chapter 8 contains an extended example of the SAME. Chapter 9 describes the use of the SAME in applications which use dynamic SQL or Ada multi-tasking.

The SAME is supported by the SAME standard packages. A complete listing of these package specifications, along with suggested package bodies, appears in Appendix A. There are also appendices containing a quick reference guide and a glossary of terms.

```

with SQL_Standard, Example_Module, Example_Definitions,
     SQL_Communications_Pkg, SQL_Database_Error_Pkg;
use SQL_Standard;
package body Example_Interface is

package ExMod renames Example_Module;
package SCP renames SQL_Communications_Pkg;
package SDEP renames SQL_Database_Error_Pkg;
package ExDef renames Example_Definitions;

procedure Open (Lower_Bound : Status_Not_Null) is
begin
    ExMod.X_Open(Int(Lower_Bound), SCP.SQLCODE);
    If SCP.SQLCODE in SQL_Error then
        SDEP.Process_Database_Error;
        raise SCP.SQL_Database_Error;
    end if;
end Open;

procedure Fetch (Tuple : in OUT Part_Nbr_City_Pairs;
                 Found : OUT Boolean) is
City_Buf : Char (ExDef.CITY_Not_Null'Range);
City_Indic : Indicator_Type;
begin
    ExMod.X_Fetch(Char(tuple.Pno), City_buf, City_Indic, SCP.SQLCODE);
    case SCP.SQLCODE is
        when Not_Found =>
            Found := false;
        when SQL_Error =>
            SDEP.Process_Database_Error;
            raise SCP.SQL_Database_Error;
        when 0 =>
            If City_Indic < 0 then
                assign(tuple.City, Null_SQL_Char);
            else
                assign(tuple.City,
                    City_Ops.With_Null(City_Not_Null(City_Buf)));
            end if;
            Found := true;
        when others => null; -- standard has no such codes
    end case;
end Fetch;

procedure Close is
begin
    ExModX_Close(SCP.SQLCODE);
    if SCP.SQLCODE in SQL_Error then
        SDEP.Process_Database_Error;
        raise SCP.SQL_Database_Error;
    end if;
end Close;

end Example_Interface;

```

Figure 1-13: Body of the Abstract Interface -- The Abstract Module

2. The SAME Typing Model

This section describes the model of data typing employed by the SAME. The model's objective is to integrate the data semantics of Ada and SQL to the extent that is desirable and practicable. The problems to be solved in such an integration are:

- **The differences between the typing models of Ada and SQL.** SQL offers a limited set of primitive data types. It does not offer a mechanism for user-defined types. The abstract typing mechanisms of Ada are a central aspect of the language. An Ada program prefers a view of the database contents consistent with a set of abstract, application-oriented types.
- **The null value.** SQL provides a means of processing missing or incomplete information. This is the null value and three-valued logic. These notions do not appear in Ada.
- **String processing.** Ada and SQL give subtly different semantics to the string comparison operators. Further, the Ada predefined type string is by definition a sequence of ASCII characters. SQL strings are over an implementor-defined character set.
- **Decimal fixed point arithmetic.** Ada fixed point arithmetic does not resemble SQL decimal arithmetic. More importantly, Ada compilers do not recognize the machine-specific packed decimal formats in which SQL database management systems store decimal data.
- **Non-standard data types.** Many database management systems recognize data types not in the ANSI standard. The date-time data type is an example of this. Ada programmers may wish to store enumeration types in SQL databases, even though SQL does not recognize such types.

The SAME solution to these problems aims at good performance in both time and space. It achieves a direct mapping between SQL and Ada types [11] which requires no data conversions. Each bit pattern representing a non-null value of a database column represents the same value of the Ada data type which describes it.⁹

The SAME typing model is flexible. An overview of it is given in Figure 2-1. At the lowest or concrete level of the interface, at which the calls to the concrete DBMS module appear, database values are described by Ada types designed in conformance with SQL requirements. These types are reviewed in the next subsection. Except for Chapter 7, these guidelines assume a compiler for the module language conforming to the recommendations in [16] which are incorporated in [5]. In Chapter 7, techniques are presented for low cost implementations of SAME in environments without a module compiler.

As shown in Figure 2-1, the concrete types at the concrete level are transformed into abstract types at the abstract level. The three branches of that diagram represent three different treatments of data semantics.

⁹The Ada application program sees the database through a set of abstract, application-oriented types. These types and their derivation are described in Chapter 3. This section is concerned with the concrete representation of database values.

- **Ada semantics.** Each database column is represented by an Ada type whose arithmetic, comparison, and assignment operations are those of Ada. With these semantics, treatment of database and non-database data is uniform throughout the Ada program.
- **SQL semantics.** Each database column is represented by an Ada type whose arithmetic, comparison, and assignment operations simulate those of SQL. With these semantics, treatment of database data is uniform between the SQL and Ada portions of the complete application.
- **User-defined semantics.** Each database column is represented by an Ada type whose arithmetic, comparison, and assignment operations are user defined. This treatment allows for user extensions of the method.

The choice of treatment is the responsibility of the application designer. This section describes the realization of those semantic treatments.

As mentioned, the next section reviews the concrete treatment of SQL data. It is this treatment which achieves the direct mapping mentioned earlier. Chapter 3 describes the development of the abstract domains. Section 3.1 discusses the treatment of null values in the SAME and how that affects application programs. Section 3.3 continues that discussion, showing how the abstract types implementing SQL semantics can be arranged into type hierarchies and Ada range constraints can be simulated for them. Section 3.4 gives the additional information needed to understand SQL strings and their SAME implementation. Section 3.5 explains the SAME simulation of SQL decimal fixed point arithmetic and Section 3.6 describes the treatment of data types not covered in the ANSI standard. An implementation of a date-time data type and implementations of support for SQL storage of Ada enumeration types are presented in Section 3.6. The section serves as a model for user extensions to the SAME typing model.

The sections described above each deal with individual columns in isolation. Section 3.7 puts the results of those sections together into a description of the database.

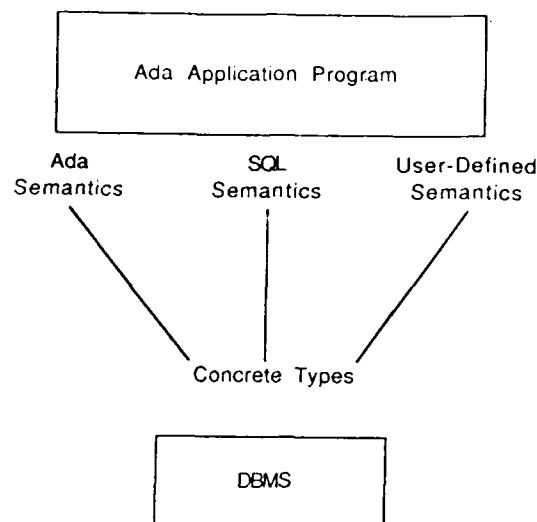


Figure 2-1: The SAME Typing Model

2.1. Concrete Types

At the lowest, or concrete, level of the SAME SQL Ada interface, the level at which the calls to concrete module routines appear, all parameters have types which appear in the package `SQL_STANDARD`. This package was created by the SAME Design Committee (SAME-DC) as a recommended change to the ANSI SQL interface to Ada [16] [3].¹⁰ A listing of this package appears in Figure 2-2. Each type definition in `SQL_STANDARD` directly defines the SQL type with the same name.¹¹ The definition is direct in the sense used previously: the value sets underlying the types in `SQL_STANDARD` are exactly the value sets underlying the corresponding SQL types. Further, under reasonable assumptions,¹² the data encodings will be identical and no data conversion will be necessary.

All of this is achieved by judicious choice of the implementor-defined values in `SQL_STANDARD`. These values are specific to the database management system in use. Once they have been determined, the package will be compiled as part of the installation procedures for the SAME standard packages into an Ada library within which it may be referenced by other SAME standard packages. Application programmers need not be concerned with this package; application programs do not reference it.

¹⁰These recommendations were accepted by the responsible ANSI subcommittee and appear in their current proposal for Ada support in SQL [5].

¹¹Although `SQLCODE_TYPE` is not a type defined in SQL, `SQLCODE` acts as though it were a type as well as a variable in [2] and [5].

¹²The assumptions are that the DBMS, at the application programming interface, delivers numeric values in the encoding of the machine and that the Ada compiler uses these encodings as well. This should be true in almost every case.

```

package sql_standard is
package Character_Set renames csp:
  subtype Character_Type is Character_Set.cst;
  type Char is array (positive range <>)
    of Character_Type;
  type Smallint is range bs..ts;
  type Int is range bi..ti;
  type Real is digits dr;
  type Double_Precision is digits dd;
-- type Decimal is to be determined;
  type Sqlcode_Type is range bsc..tsc;
  subtype Sql_Error is Sqlcode_Type
    range Sqlcode_Type'FIRST .. -1;
  subtype Not_Found is Sqlcode_Type
    range 100..100;
  subtype Indicator_Type is t;

--   csp is an implementor-defined package and cst is an
--   implementor-defined character type. bs, ts, bi, ti, dr, dd, bsc,
--   and tsc are implementor defined integral values. t is int or
--   smallint corresponding to an implementor-defined <exact
--   numeric type> of indicator parameters.

end sql_standard;

```

Figure 2-2: The Package SQL_STANDARD

The values appropriate to the definition of the integer and floating point types will generally be easily available in the DBMS documentation. Likewise the definition of SQLCODE_TYPE should not be difficult. (It is likely to be identical to one of the integer types.) The floating point types will also be defined in the DBMS documentation. It may also be necessary to examine the documentation for the Ada compiler, particularly true for the values of System.Max_Int and System.Max_Digits.

The treatment of character data in SQL_STANDARD is meant to allow for non-ASCII data. The type CHAR is defined on analogy to the Ada predefined type STRING but with respect to a character type which can be specified by the implementor. To use these definitions with ASCII strings, set *csp* to STANDARD and *cst* to CHARACTER.

The subtypes SQL_ERROR and NOT_FOUND of SQLCODE_TYPE are provided for the benefit of programmers, such as authors of abstract modules, who write their own error detection routines. For example, one may write

```

  if SQLCODE is in Sql_Error . . .
or
  case SQLCODE is
    when 0 =>
      -- error free return
    when Not_Found =>
      -- no record found
    when Sql_Error =>
      -- error condition from DBMS
    when others =>
      -- standard describes no such codes
  end case;

```

For more on the SAME treatment of exceptional conditions, see Chapter 4.

The SAME standard packages also depend upon the package SQL_SYSTEM (see Figure 2-3) which defines two constants, the values of which cannot be deduced from SQL_STANDARD. The constant MAXCHRLen is the length of the longest character string supported by the DBMS. The constant MAXERRLEN is the length of the longest error message returned by the DBMS-supplied error message function. See Chapter 4 for details.

```
-- SQL_System is a "platform-specific" package
-- within the SAME
package SQL_System is
  -- MAXCHRLen is the upper bound of the SQL_Char_Pkg
  -- subtypes SQL_Char_Length and SQL_Unpadded_Length
  -- SQL_Char_Length is a subtype of Natural with a lower bound
  -- of 1
  -- SQL_Unpadded_Length is a subtype of Natural with a lower
  -- bound of 0

  MAXCHRLen : constant := str_len;

  -- MAXERRLEN is the maximum length of the error message
  -- string returned from the DBMS error message function

  MAXERRLEN : constant := msg_len;

end SQL_System;
```

Figure 2-3: The Package SQL_System

Creation and compilation of the SQL_STANDARD and SQL_SYSTEM package specifications are part of the installation of the SAME standard packages. The installation guide for the SAME standard packages [14] contains details of the installation process.

3. Developing the Abstract Domains

The types in `SQL_STANDARD` define the representation of SQL data to the Ada compiler. As illustrated in Section 1.2, applications developed using the SAME method view the database through a collection of abstract domains. These abstract domains are built on top of type definitions provided in the SAME standard packages or in similar packages defined by the user (see Section 3.6).

There exists a support package in the SAME standard packages for each of the types in `SQL_STANDARD` (except for `SQLCODE_TYPE`). The package `SQL_INT_PKG` gives support to abstract types based on the SQL `Int` type; `SQL_CHAR_PKG` supports character strings, etc.

Each of these packages defines two types. One of these types is a visible Ada type derived from the corresponding type in `SQL_STANDARD` with no added constraints. These type names are formed from the package name by dropping the `_Pkg` suffix and appending the suffix `_Not_Null`. Thus, `SQL_Int_Not_Null` is defined in the package `SQL_INT_PKG` as `new SQL_Standard.Int`; `SQL_Char_Not_Null` is defined in `SQL_CHAR_PKG` as `new SQL_Standard.Char`, etc.

The second type defined in each package is a limited private type. These type names are formed by dropping the `_Pkg` suffix and adding no additional suffix. Thus `SQL_INT_PKG` defines `SQL_Int`, `SQL_CHAR_PKG` defines `SQL_Char`, etc. These limited private types are used to support SQL data semantics. In particular, objects of these types can take on the SQL null value,¹³ whereas objects of the `_Not_Null` types cannot.

As is shown in the introduction, an abstract domain is represented in the SAME by two type definitions derived, directly or indirectly, from the types in a support package. (Character string types further require two subtype definitions. These are explained in Section 3.4, below.) Conventionally, the name of the type derived from the `_Not_Null` type retains the `_Not_Null` suffix; the name for the type derived from the limited private type appends the suffix `_Type`; these derivations and naming conventions are illustrated in Figure 1-7. The types are referred to in this document as the *visible Ada*, or *_Not_Null* type, and the *limited private*, or *_Type* type.

The creation of the abstract domain definitions completes the first step in the description of the database within the SAME method. The second step consists of collecting the definitions into Ada package specifications. These are called *domain packages* and their formation is defined in Section 3.7.

3.1. The SAME Treatment of SQL Null Values

Objects or values that are directly or indirectly database values are to be stored as objects of one of the types making up an abstract domain definition. In cases in which it is possible for these database values to take on the SQL null value, they must be stored as values of the limited, `_Type` type. In cases in which it is logically certain that a value cannot be null, the visible `_Not_Null` type can be used. This logical certainty can be supplied either by SQL

¹³The SQL null value should not be confused with the null value of an access type.

or by the application logic. The data definition facilities of SQL can restrict the value of a table column to exclude the null value; the data manipulation statements of SQL can filter out rows with null values in specified columns. Within an application, it may be logically certain that null values have been previously filtered out. If the absence of the null value is not logically certain in this sense, then the limited type must be used. The SAME standard packages are defined in such a way as to guarantee a runtime error, namely, the exception `Null_Value_Error`, if a null value is inadvertently used as though it were not null.

Consider, then, a situation in which the null value is logically possible and a given object has one of the SAME limited types. As part of its method, the SAME offers three treatments of these objects. These treatments are coding disciplines enforced on application programmers. The SAME allows these treatments to be intermixed in an application program in any way, subject only to whatever local standards and guidelines may exist.

3.1.1. The Minimalist Approach

In the minimalist approach, objects of limited types are treated solely as value repositories. All manipulation of and access to the values of these objects is done by first extracting the value from the limited object into an object of the corresponding visible or `_Not_Null` type. An advantage of this approach is that, as the `_Not_Null` types are visible Ada types, the predefined Ada operations may be used on objects of those types. Furthermore, as objects of those types may not be null, it is unnecessary to check for the null value when accessing such objects. The minimalist approach may result in marginal runtime reductions. More importantly, the minimalist approach may appear more natural to some programmers.

Each of the SAME standard packages offer two sets of functions to support the minimalist approach. They are testing functions and conversion or extraction functions. The extraction functions will raise the `Null_Value_Error` exception if applied to an object whose value is null.

- **Testing functions.** These are the Boolean-valued functions `Is_Null` and `Not_Null`. These functions are declared in the specification of the appropriate SAME standard packages (`SQL_Int_Pkg`, etc.) in which the limited type and visible types are also declared. Therefore, when the pair of types defining an abstract domain are derived from those types, these subprograms are derived for the new type.
- **Conversion functions.** These are the functions `With_Null` and `Without_Null`.¹⁴ The function `With_Null` takes an object of the visible `_Not_Null` type and returns a non-null object of the corresponding limited, `_Type` type. The function `Without_Null` takes an object of the limited type and returns an object of the `_Not_Null` type. `Without_Null` raises the exception `Null_Value_Error` if its input is the null value.

¹⁴The character string support provided by `SQL_Char_Pkg` includes other conversion functions. They are described in Section 3.4.

Example

Consider the following fragment of application logic, referencing the Parts - Supplier database of the introduction. Suppose there exist two variables, `City`, of type `City_Type` (a derived type of `SQL_Char_Pkg.SQL_Char`), and `Quantity`, of type `Quantity_Type` (derived from `SQL_Int_Pkg.SQL_Int`). In other words, each of the variables may have the null value. We need to write a code fragment which increments a counter if the value of `City` is "Pittsburgh" or the value of `Quantity` exceeds 1000. Furthermore, we want to keep a running total of the `Quantity` values from rows which qualify in this way. Omitting variable declarations for the sake of brevity, we have the following code fragment (the variable `Sum_Quantity` has type `Quantity_Not_Null`):

```
    If (Not_Null(City) and then Without_Null(City) ="Pittsburgh")
      or else
        (Not_Null(Quantity) and then Without_Null(Quantity) > 1000)
      then
        Counter := Counter + 1;
        If Not_Null(Quantity) then
          Sum_Quantity := Without_Null(Quantity) + Sum_Quantity;
        end if;
      end if;
```

3.1.2. The Full SQL Approach

An alternative to the minimalist approach to null values is the "full SQL" approach. Using this approach, objects of the `_Type` types are accessed and manipulated directly, without having to be extracted or converted to a visible Ada type. To enable this approach, the SAME standard packages declare overloaded versions of the standard Ada arithmetic and comparison operators. These versions extend the semantics of those operators to include the null value. The null value is processed according to the rules of SQL. An application using this approach treats database data in a uniform way in the Ada and SQL portions of the application. To use the approach, it is necessary to understand how SQL processes the null value.

SQL defines arithmetic and comparison operators for sets including the null value. The semantics are as follows:

- **Arithmetic:** Any arithmetic operation applied to a null value results in the null value; otherwise, the operation is defined to be the same as the Ada operation for the integer and floating point types. (See also Section 3.5 for decimal arithmetic.)
- **Comparison:** The comparison of any value to the null value results in a new truth value called `UNKNOWN`; otherwise the operation is defined as in Ada for the integer and floating point types. (See Section 3.4 for the string comparisons.)

The overloaded operators provided by the SAME standard packages implement these semantics. The comparison operators, `Equals`, `Not_Equals`, `<`, `<=`, etc., return objects of type `Boolean_With_Unknown`. This is an Ada enumeration type with value set (`FALSE`, `UNKNOWN`, `TRUE`). The SAME standard package `SQL_Boolean_Pkg` contains declaration of the Boolean functions **and**, **or**, **not** and **xor** defined on this type which implement the three-valued logic of SQL. The definitions of these functions in three-valued logic are given by the truth table in Figure 3-1.

A	B	A and B	A or B	A xor B	not A
T	T	T	T	F	F
T	F	F	T	T	F
F	F	F	F	F	T
T	U	U	T	U	F
F	U	F	U	U	T
U	U	U	U	U	U

T - true F - false U - unknown
 Rows not shown follow by symmetry

Figure 3-1: Three-Valued Logic

Example

The prior example concerning Cities and Quantities can be recoded as

```
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
...
if Is_True(Equals(City, With_Null("Pittsburgh"))) or
           Quantity > With_Null(1000)) then
  Counter := Counter + 1;
  if Not_Null(Quantity) then
    assign(Sum_Quantity, Quantity + Sum_Quantity);
  end if;
end if;
```

This encoding is functionally equivalent to the prior encoding. The Counter will be incremented under the same circumstances as before; namely, when at least one of City or Quantity has the proper value. This encoding illustrates mixed usage of the two treatments. The final value of Sum_Quantity, now of type Quantity_Type, will be the sum of all non-null quantities encountered. Had the test for the null value not been present, and had a null value been encountered, the result would be the null value. This treatment of summing is equivalent to the SQL SUM set function which also sums columns of data after filtering out null values.

3.1.3. A Compromise Approach for Comparison Operators

This section considers only the comparison operators, e. g., =, >, >=, etc., and offers a third alternative to their use. One of the difficulties with the comparison operators described in Chapter 3.1.2 is that the values they return are not of the predefined type Boolean. This means that predicates formed with these operators cannot appear as the condition of an if statement unless they are first converted to Boolean using one of the functions Is_True, Is_False or Is_Unknown defined in SQL_Boolean_Pkg, as was shown in the prior example. Further, since the rules of Ada require that any overloading of the equality operator "=" return Boolean, the three-valued equality comparison function must be coded as the prefix function Equals and its complement as the prefix function Not_Equals. Finally, it is reasonable to assume that the most frequently used function to cast a value of type Boolean_with_Unknown to type Boolean is the Is_True function used in the prior example. Indeed, the semantics of the SQL WHERE clause are precisely evaluation using the rules of Section 3.1.2 followed by an application of Is_True.

For the reasons given in the prior paragraph, sets of overloads of the comparison operators are defined in the support packages on the null bearing `_Type` types. These overloads return Boolean, not Boolean_with_Unknown as the operators of Section 3.1.2. These overloads are defined as follows: for the operator "op," and objects O_1 and O_2 of a null bearing `_Type` type, the Boolean-valued expression

O_1 op O_2
is defined as

```
Is_True( $O_1$  op  $O_2$ )
```

where in the second expression, "op" is the overloading which returns Boolean_with_Unknown. (If "op" is "=" or "/=", the second expression is written in prefix notation, using `Equals` or `Not_Equals`, respectively.) If P is any Boolean combination of comparisons from this section, and P' is the result of substituting the three-valued operators from Section 3.1.2 into P, then the value of P is `Is_True(P')`.

Example

The running example of this section can also be coded as

```
if City = With_Null("Pittsburgh") or else
    Quantity > With_Null(1000) then
    Counter := Counter + 1;
    if Not_Null(Quantity) then
        assign(Sum_Quantity, Quantity + Sum_Quantity);
    end if;
end if;
```

A Note on Type Ambiguities

Notice that the context determines whether a given operator is three-valued or Boolean valued. If the predicate P does not contain the equality operator, then the predicate P' as defined above is syntactically identical to P. The context must be sufficient to determine which interpretation is meant. For example, the context `Is_True(.)` is sufficient to determine that the three-valued interpretation is required for P' in `Is_True(P')`. Similarly, the context of P in `if P then ... end if;` is sufficient to determine that P is Boolean valued. Consider the expression $O_1 > O_2$, which has both a three-valued and Boolean interpretation. The case statements

```
case Boolean' ( $O_1 > O_2$ ) is
    when TRUE => ... ;
    when FALSE => ...;
end case;

case Boolean_with_Unknown' ( $O_1 > O_2$ ) is
    when TRUE => ... ;
    when FALSE => ...;
    when UNKNOWN => ...;
end case;
```

would not compile were the type qualifications not present. As written, these statements will perform as expected.

The presence of an equality operator or a Boolean short circuit control form within a predicate is sufficient to determine its type. Therefore the predicate

`Equals(o1, o2) or o1 > o2`

is unambiguously of type `Boolean_with_Unknown` and the expression

`o1 = o2 or o1 > o2`

is unambiguously of type `Boolean`; whereas the expression

`o1 >= o2`

is ambiguous. Similarly, the expression

`(o1 >= o2) or (o1 >= o3)`

is ambiguous, but the expression `(O1 >= O2) or else (O1 >= O3)` is unambiguously of type `Boolean`.

A Note on Logic

The Boolean-valued comparison operators discussed in this section do not obey all the normal rules of propositional logic. Furthermore, due to the definition of Ada, their behavior is inconsistent. The problem arises in the so-called rule of double negation.

Again, let *P* be any predicate formed using the Boolean operators **and** and **or** from Boolean-valued expressions. Now let *P'* represent the result of performing the following substitutions to *P*:¹⁵

- each comparison operator is replaced by its negation; that is, = is replaced by /=, < is replaced by >=, etc.
- **and** is replaced by **or**
- **or** is replaced by **and**

This substitution produces the result of taking the expression **not** *P* and distributing the negation over the other operators. The rule of double negation states that the equality

$P = \text{not } P'$

is valid, that is, always holds. This rule does not apply to predicates formed from the Boolean-valued comparison operators of this section.¹⁶ This fact can be used to advantage. For example, in the statement

```
if Quantity > With_Null(1000) then
  ...
end if;
```

the sequence of statement in the **then** clause are executed only for non-null quantities in excess of one thousand. In contrast, the statements in the **then** clause of

```
if not Quantity <= With_Null(1000) then
  ...
end if;
```

¹⁵The Boolean operators **not** and **xor** have been omitted to simplify the substitution. Given that the negation of every comparison operator is a comparison operator, as in the first bullet item, any predicate using **not** and **xor** can be recoded as one using **and** and **or** exclusively.

¹⁶The law of double negation is usually stated as the equality $P = \text{not}(\text{not } P)$. This law does hold to predicates formed from the operators of this section.

will be executed for those quantities and also for all null quantities. Recall that the null value in SQL represents missing information. The null *Quantity* represents a fixed but unknown value for *Quantity* which may exceed one thousand. Thus the second if statement, which is often called the maximal solution, executes the sequence of statements in the **then** clause for any quantity which might fit the predicate, while the first statement, the minimal solution, executes that sequence only for quantities which necessarily do fit the predicate.

Regrettably, this behavior is not consistent. The inconsistency stems from the fact that Ada does not allow an overloading of the inequality operator **"/="** to be independently defined. Rather, **"/="** is implicitly defined to be the complement of **"="**. In short, the equivalence

$$(o_1 = o_2) = \text{not } (o_1 \neq o_2)$$

is valid. When a complex predicate contains both **"="** (or **"/="**) and other comparison operators, the result of the double negation process outlined above is difficult to predict. In such cases it is best to use the three-valued operators and the **case** statement. Thus the maximal solution to the running example of these sections can be written as

```
case Boolean_with_Unknown' (  
    Equals(City, With_Null("Pittsburgh"))  
    or  
    Quantity > With_Null(1000)) is  
    when TRUE | UNKNOWN => <as before>  
    when FALSE => null;  
end case;
```

The extended example in Chapter 8 contains further discussion of these details.

3.2. The Image and Value Functions

In addition to the testing, conversion, comparison, and arithmetic functions types and assignment procedure, the SAME support for integer in the packages `SQL_Int_Pkg` and `SQL_Smallint_Pkg` includes the functions `Image` and `Value`. These functions are semantically identical to the Ada attribute functions `'Image` and `'Value` except that they operate on character strings of type `SQL_Char` (or `SQL_Char_Not_Null`) rather than the predefined type string. This allows character set independent programs to be written, as strings of these types are always over the machine's native character set. When used with objects of some `_Not_Null` type, these functions take or return strings of type `SQL_Char_Not_Null`; when used with an object of a null bearing `_Type` type, they take or return `SQL_Char` strings, with the null value of the source type being transformed into the null value of the target type. Notice that the character string operands of these functions are of the base types declared in `SQL_Char_Pkg`. Application programs do not have visibility to that package. A means of getting visibility to the base types is given in Section 3.8.

3.3. Range Constraints and the Generic Sub-Packages

Many relational database management systems provide for data integrity constraints.¹⁷ Among these there is usually the ability to apply range constraints to numeric columns. The SAME extends this ability to Ada program variables holding database values.

Example

Suppose all status values must be positive. In that case, the definitions of the abstract Status domain would be

```
type Status_Not_Null is new SQL_Int_Not_Null
    range 1 .. SQL_Int_Not_Null'LAST;
type Status_Type is new SQL_Int;
package Status_Ops is new
    SQL_Int_Ops (Status_Type, Status_Not_Null);
```

Notice that the range constraint is applied to `_Not_Null` type only. `Status_Type` is a limited private record type, to which range constraints cannot be applied. The generic instantiation `Status_Ops` creates an `Assign` procedure which will enforce the range constraint on objects of `Status_Type`.

The specification of the package `SQL_Int_Ops`, which appears within the specification of the package `SQL_Int_Pkg`, is given in Figure 3-2. The packages `SQL_Smallint_Ops`, `SQL_Real_Ops` and `SQL_Double_Precision_Ops` are identical to `SQL_Int_Ops`, with the obvious modifications. `SQL_Char_Ops` is slightly different and is described in Section 3.4.

Notice that the generic takes two formal parameters which are types and three which are subprograms. The subprograms will default to subprograms with the appropriate names and profiles, which are derived by the type derivation. (The packages should be instantiated in the declarative region in which the derived types are declared. See Section 3.7.) Therefore, when instantiating these packages, only the types should be passed as actuals.

Notice that the generic subpackage generates three subprograms which provide conversion and assignment procedures. It is not necessary to generate the arithmetic and comparison operators. They are derived with the derivation of the type `Status_Type`.

The procedure `Assign` produced by the generic instantiation implements range constrained assignment for the limited private types. It does this by calling the procedure `Assign_With_Check`¹⁸ and passing it the values of the attributes `'FIRST` and `'LAST` from the `_Not_Null` type. See the appendix for the complete code.

Note: The implementation of range constraints by the SAME standard packages is meant to support the implementation of range constraints by the DBMS. As this feature is missing from the current SQL standard, a given DBMS may not support it. This does not mean that range constraints cannot be used in Ada applications employing the SAME. The constraint

¹⁷These constraints do not appear in the current ANSI standard [2] but do appear in the follow-on standard in development [4].

¹⁸This procedure is not meant to be called directly by application programs. Applications should use only the `Assign` function produced by the generic instantiation.

```

generic
  type With_Null_Type is limited private;
  -- derived from Sql_Int
  type Without_Null_Type is range <>;
  -- derived from Sql_Int_Not_Null;
  -- for floating point types
  -- range is replaced with digits

  with function With_Null_Base(Value : Sql_Int_Not_Null)
    return With_Null_Type is <>;
  with function Without_Null_Base(Value : With_Null_Type)
    return Sql_Int_Not_Null is <>;
  with procedure Assign_With_Check (
    Left : in Out With_Null_Type; Right : With_Null_Type;
    First, Last : Sql_Int_Not_Null) is <>;
-- subprograms with the above names
-- appear in Sql_Int_Pkg specification

package Sql_Int_Ops is
  function With_Null (Value : Without_Null_Type)
    return With_Null_Type;
  function Without_Null (Value : With_Null_Type)
    return Without_Null_Type;
  procedure Assign (Left : in Out With_Null_Type;
    Right : With_Null_Type);
end Sql_Int_Ops;

```

Figure 3-2: The Generic Subpackage Sql_Int_Ops

"all status values are positive," if applied in the SAME abstract domain definitions as described above, should represent a constraint on the real world. If this constraint is true of the real world, then any non-positive value of Status is invalid and represents a corruption of the database. If this constraint is not supported by the DBMS, the exception Constraint_Error will be raised when this database corruption is encountered. That may cause the abnormal termination of one database application due to the improper behavior of a different application, that application which inserted the invalid data. The incorrect application could not have been written in Ada using the SAME.

The conversion functions With_Null and Without_Null are also generated by the _Ops generic subpackages. These functions convert between the two types making up an abstract domain. Ada subprogram derivation rules will not generate functions with these parameter profiles.

The _Ops generic subpackages were designed to reduce compile-time and runtime space utilization. Only those subprograms that could not be derived using Ada subprogram derivation rules are instantiated using generic instantiation.

A Note on Type Derivation and Subtyping

The abstract domains defining the database in Ada can be arranged into type and subtype hierarchies in the usual way. For example, suppose it is desirable to define preferred suppliers as those suppliers having a status greater than 100. This can be captured in subtype declarations as follows.

```

subtype Preferred_Status_Not_Null
  is Status_Not_Null range 101 .. Status_Not_Null'LAST;
subtype Preferred_Status_Type is Status_Type;
package Preferred_Status_Ops is new SQL_Int_Ops
  (Preferred_Status_Type, Preferred_Status_Not_Null);

```

However, care must be exercised in naming the subprograms operating on variables of the subtype. The subprograms generated in the package Preferred_Status_Ops have the same parameter profiles as those generated in the package Status_Ops defined in Figure 1-7. This is because parameter profiles depend only on base types, not on subtypes. Consider the following program fragment.

```
Preferred_Status_Variable : Preferred_Status_Type;
. . .
begin
. . .
    Status_Ops.assign(Preferred_Status_Variable,
                      Status_Ops.with_Null(1));
. . .
end;
```

This will execute without raising an exception and will result in the variable's having a value out of range. Further, the subprogram declarations in the packages Status_Ops and Preferred_Status_Ops hide each other, if both are brought into scope with use clauses.

Warning: Since range constraint checking of objects of the null bearing `_Type` types is done by the generated Assign procedures and not directly by the compiler, these constraints do not behave exactly like Ada constraints. In particular, if an arithmetic expression resulting in a `_Type` object is passed as an actual parameter to a procedure, it will not be range constrained and may not satisfy the range constraint. For safety, assign the expression to a temporary variable of the `_Type` and pass the temporary as the actual.

3.4. Character Data

The SAME treatment of character string data is similar to its treatment of integer and floating point data. Each abstract character string domain is represented by two type declarations. One of the types is a visible Ada type; the other is a limited private type with operations defined on it that simulate the corresponding SQL operations. Character string variables and database columns do not have associated range constraints, but they do have lengths. The length of an SQL character string column is part of its definition. Abstract domain definitions for character string domains also contain a length.

The SQL semantics of character data include the semantics of the null value for strings¹⁹ as described in Section 3.1.2. Unlike the case of integer and floating point data, for which operations on non-null values have the same effect in Ada and SQL, SQL's definition of assignment and comparison for character strings differs from Ada's definition. For example, when comparing two strings, SQL pads the shorter string with blanks (*Database Language-SQL*, paragraph 5.11.5 [2]).

The comparison of two character strings is determined by the comparison of `<character>s` with the same ordinal position. If the strings do not have the same length, then the comparison is made with a working copy of the shorter string that has been effectively extended on the right with `<space>s` so that it has the same length as the other string.

¹⁹The null string value is distinct from the null string, i.e., the string of length 0.

Very similar behavior governs the assignment of character strings to database columns in SQL INSERT and UPDATE commands (cf. *Database Language—SQL*, general rule 7.b of Sections 8.7, 8.11 and 8.12 [2]).

The SAME standard package SQL_Char_Pkg defines the type SQL_Char_Not_Null as a derived type of SQL_Standard.Char (see Figure 2-2) with no added constraints. SQL_Char_Not_Null is therefore an unconstrained one dimensional array whose component type is specified when SQL_STANDARD is compiled. SQL_Char_Pkg also declares a limited private, discriminated record type SQL_Char and comparison and assignment operations on that type which simulate the SQL operations. The discriminant is named Length and is of type SQL_Char_Length, a subtype of INTEGER declared in SQL_Char_Pkg. The discriminant value is used to specify the character string length.

SQL_Char_Pkg also contains a generic subpackage, SQL_Char_Ops. As before, it generates conversion functions between a type derived from SQL_Char_Not_Null type and a type derived from SQL_Char. Together the two type definitions make up the abstract domain definition. (There is no need for the generic subpackage to create an Assign procedure. The version derived by the derived type declaration will suffice.) Notice, however, that the _Not_Null type is not the Ada predefined type, string. Rather, the _Not_Null type is a derived type of SQL_Char_Not_Null, itself a derived type of SQL_Standard.Char. That type may or may not be a renaming of the predefined type string (that is, Standard.string), as the DBMS character set may or may not be ASCII. SQL_Char_Pkg exports functions which convert between each of the _Not_Null and the limited private type and the predefined type string. These functions will perform character set conversions if necessary. (The identity of the character set conversion function is set during SAME installation. See the installation guide [14] for more details.)

The remainder of this section is as follows. The generic subpackage is displayed and explained. Abstract domain definitions for character data, which differ slightly from the integer and floating point case, are then described and explained. The functions which convert to and from the predefined string type are then explained. Finally, a function for extracting substrings from character strings of the limited private type and an operator for concatenating two such strings are described.

The specification of the generic subpackage SQL_Char_Ops appears in Figure 3-3. This generic subpackage is to be instantiated in the same manner as the integer and floating point subpackages: only the types are passed as actuals, the formal subprograms are meant to default.

The functions With_Null and Without_Null generated by instantiation of this package have the same intended meaning as before: to convert between the two types of an abstract domain. The function Without_Null_Unpadded returns the value of its input with trailing blanks removed; the last character in the result of this function is never blank. If the input string is all blank, the output is an array of length zero. SQL_Char_Pkg exports the function Unpadded_Length with operand SQL_Char and result type SQL_Unpadded_Length, a subtype of NATURAL. The defining property of the function is

```

generic
type With_Null_Type is limited private;
  -- derived from SQL_Char
type Without_Null_Type is array (positive range <>)
of SQL_STANDARD.Character_Type;
  -- derived from SQL_Char_Not_Null
with function With_Null_Base (Value: SQL_Char_Not_Null)
return With_Null_Type is <>;
with function Without_Null_Base (Value: With_Null_Type)
return SQL_Char_Not_Null is <>;
with function Without_Null_Unpadded_Base (Value: With_Null_Type)
return SQL_Char_Not_Null is <>;
package SQL_Char_Ops is
function With_Null (Value : Without_Null_Type)
return With_Null_Type;
function Without_Null (Value : With_Null_Type)
return Without_Null_Type;
function Without_Null_Unpadded (Value : With_Null_Type)
return Without_Null_Type;
end SQL_Char_Ops;

```

Figure 3-3: The Generic Subpackage SQL_Char_Ops


```
Without_Null_Unpadded(x)'LENGTH = Unpadded_Length(x)
```

Notice that (assuming x is not the null value)

```
Without_Null(x)'LENGTH = x.LENGTH
```

It should be noted that `Without_Null`, `Without_Null_Unpadded`, and `Unpadded_Length` raise `Null_Value_Error` when given the null value as input.

The generic `SQL_Char_Ops` explains to some extent the structure of abstract domain definitions for character data. A character string abstract domain definition contains two type declarations and two subtype declarations, along with the instantiation of the generic subpackage. The following declaration of the abstract domain `PNO` is copied from Figure 1-7.

```
type PNONN_Base is new SQL_Char_Pkg.SQL_Char_Not_Null;  
subtype PNO_Not_Null is PNONN_Base (1..5);  
type PNO_Base is new SQL_Char_Pkg.SQL_Char;  
subtype PNO_Type is PNO_Base (PNO_Not_Null'Length);  
package PNO_Ops is new  
    SQL_Char_Pkg.SQL_Char_Ops(PNO_Base, PNONN_Base);
```

The type definitions, whose type names have the suffix `_Base`, declare unconstrained types. The subtypes complete the domain definition by supplying the string length. The subtype declarations are to be used in declaring variables of the abstract domain. Thus the subtype declarations have the suffixes `_Not_Null` and `_Type` as appropriate.

The pattern of the above example should always be followed in the definition of character string abstract domains. The length of the character strings as they are stored in the database should be encoded as an index constraint on the `_Not_Null` subtype. The value of the discriminant in the definition of the `_Type` subtype is the `Length` attribute value of the `_Not_Null` subtype. This pattern guarantees that the `_Type` and `_Not_Null` subtypes are consistent.

The formal type parameter `Without_Null_Type` of the generic package `SQL_Char_Ops` (see Figure 3-3), is an unconstrained array type. Therefore, the actual type parameter must also be unconstrained (see *LRM* [15] 12.3.4(2)). This explains the division of the declaration of the `_Not_Null` type into two pieces. Notice that, as the unconstrained types are passed to the generic instantiation, the functions it generates return objects of the unconstrained types. This is particularly important in the case of `Without_Null_Unpadded`, which returns objects whose length cannot be determined at compile time. These objects may not meet the `_Not_Null` subtype constraint, but they are valid objects of the `_Base` type. (Similar statements apply to the substring function described below.)

The functions `To_SQL_Char` and `To_SQL_Char_Not_Null`, exported by the `SQL_Char_Pkg`, take an operand of the predefined type string and return a value of either the limited private type `SQL_Char` or the one dimensional array type `SQL_Char_Not_Null` (or types making up an abstract domain definition derived from these). The length of the result is the length of the input. Both functions raise `Constraint_Error` if the input is the string of length zero.

There are two versions of the function `To_String` and `To_Unpadded_String`, one taking objects of type `SQL_Char_Not_Null` and one taking objects of type `SQL_Char` (or types derived from these). As was the case for `Without_Null` and `Without_Null_Unpadded`, the following identities hold (assuming x is of a child type of `SQL_Char` and is not null)

```
To_String(x)'LENGTH = x.Length
```

```
To_Unpadded_String(x)'LENGTH = Unpadded_Length(x)
```

and (assuming x is of a child type of SQL_Char_Not_Null)

```
To_String(x)'LENGTH = x.Length
```

There is no predefined technique for determining the length of `To_Unpadded_String(x)` if x is of a child type of SQL_Char_Not_Null.

It is impossible to reproduce exactly the syntax of the Ada slice for extracting substrings of SQL strings (strings which are objects of the type SQL_Char or a type derived from it). Therefore, there exists a function *substring* in SQL_Char_Pkg which simulates the substring function of the follow-on version of the SQL standard, SQL2 [4], in preparation. Its definition is

```
function substring (Value : SQL_Char;  
                   Start, Length : SQL_Char_Length) return SQL_Char;
```

where *substring*(str , k , m) evaluates to the substring of str starting at the k^{th} ordinal position (relative to 1) and containing m characters, unless (i) str is null, in which case *substring*(str , k , m) is also null; or (ii) $k \leq 0$ or $m < -0$ or $k+m-1 > str.LENGTH$ in which case *substring*(str , k , m) causes Constraint_Error to be raised.

SQL_Char_Pkg also exports a concatenation operator, "&", for SQL_Char. Its definition is

```
function "&" (Left, Right : SQL_Char) return SQL_Char;
```

If either operand of "&" is null, the result is null; otherwise, the result has length `Left.LENGTH + Right.LENGTH`.

3.5. Decimal Fixed Point Arithmetic

Among the data types recognized by ANSI SQL is the type Decimal. Like most of the SQL data types, the decimal type is oriented to a concrete, hardware representation. Although there is nothing in the standard that requires it, any DBMS which supports the Decimal type is likely to do so by storing values of the type in the machine's packed or binary coded decimal (BCD) representation. This section describes the support software provided by the SAME for numeric data coded in BCD.

It should be noted immediately that ANSI standard SQL as described in [2], [4], and [16] does not support decimal data in Ada programs. Therefore, this section describes SAME functionality outside of standard SQL. It may be that future versions of the ANSI standard will correct this deficiency in a manner that is not compatible with the software presented in this section. It is to be hoped that the transition to any such future standard will be relatively easy.

It is possible to read or write database values stored in decimal without any support for the type in Ada by taking advantage of SQL's weak typing. If, within an SQL statement, a decimal value is stored into or read from a parameter of some other numeric type (such as Real or Int), SQL will perform the necessary conversion automatically. The disadvantages of this approach are the time taken to do the conversion and the loss of accuracy as a result of the conversion. Decimal fractions cannot in general be accurately represented in binary notation. Furthermore, decimal representations generally allow for more digits of precision than do binary integer or floating representations. It is, as always, up to the application's designers and engineers to determine the best strategy for decimal quantities. The form of the support for BCD in the SAME is that of an abstract data type whose fundamental operations (arithmetic, comparison, etc.) are provided by assembler-level routines. It should be noted that this software is very inefficient in comparison to the software that might be produced directly by a compiler which supported BCD. As there are no such compilers at this time,²⁰ the software presented here will at least allow Ada programs access to BCD coded data.

The package SQL_Decimal_Pkg provides basic support for a non-null bearing and a null bearing type. The package defines an Ada type for BCD objects and arithmetic and comparison operators for that type. It then builds on that concrete type to provide the null bearing type with its associated operators.

²⁰No modification to the Ada language is needed to support BCD. All that is needed is an implementation of a `pragma Decimal`, which instructs the compiler to represent values of its (fixed point type) operand in BCD. Compilers are free to add such pragmas (*LRM 2.8(8)*).

3.5.1. Basic Support

The package `SQL_Decimal_Pkg` provides the Ada programmer access to the machine's BCD representation and instruction set. All of the basic operations provided by this package, arithmetic, comparison and conversion operators and functions, are implemented in assembler. Sample implementations for the VAX and IBM 360/370 instruction sets can be found in Appendix C.²¹

All of the operations are done with the maximum precision possible on the target hardware. The constant `MAX_DIGITS` defined in the specification of `SQL_Decimal_Pkg` is the number of digits in such a maximum precision number on the target machine. `SQL_Decimal_Pkg` defines an Ada type, `SQL_Decimal_Not_Null`, for Ada objects whose contents are BCD numbers of maximum precision. The type is a limited private record type with a discriminant. The component type of the record type is a fixed length array. `SQL_Decimal_Not_Null` is a limited type so as to prohibit the formation of aggregates of the type in the Ada code. This ensures that the contents of an object of the type are in valid BCD format.

The length of the array component of `SQL_Decimal_Not_Null` is calculated at compile time. The comments within the private part of the specification of `SQL_Decimal_Pkg` explain how and why the calculation is done.

The discriminant of `SQL_Decimal_Not_Null` specifies the number of scale digits, that is, digits assumed to the right of the decimal point, in objects of the type (or types derived from it). The `Assign` procedure justifies its input value around the decimal point. If a value `v1` with scale (discriminant) `s1` is assigned to an object with scale `s2`, then the value `v1` is shifted left ($s1 > s2$) or right ($s1 < s2$) as needed. In the case of a right shift, trailing digits are lost and the result is rounded. In the case of a left shift, trailing zeroes are supplied. If significant high order digits would be lost by a left shift, the exception `Constraint_Error` is raised.

The scale of the result of an arithmetic operator can be calculated as follows. For the additive operators (+, -) the result scale is the larger of the input scales. (Justification is performed automatically by the additive operators.) The result of a multiplication has scale which is the sum of the scales of its operands. The result of a division has the maximum scale possible given the values of its operands and the nature of the hardware decimal divide instruction.²² All four of the arithmetic operators raise `Constraint_Error` if the result has more significant digits to the left of the decimal point than can be accommodated. These definitions of arithmetic are modeled after the treatment given to decimal arithmetic by SQL [2].

Other noteworthy features of `SQL_Decimal_Pkg` appear in the following list. They are described with respect to the non-null bearing type `SQL_Decimal_Not_Null`. The next subsection describes the support for the null bearing type.

²¹These implementations are reentrant. Therefore, they are safe for use within Ada multi-tasking programs or other environments in which reentrancy is a requirement.

²²The VAX decimal divide instruction performs integer division on its operands and returns the quotient with the full width, i.e., precision, of the dividend. The IBM decimal divide also does integer division but returns a quotient and a remainder in the location of the dividend. Therefore a division which operates successfully on the VAX may raise `Constraint_Error` on an IBM machine.

- The parameterless functions Zero and One return the appropriate decimal constants.
- The function Shift performs multiplications by powers of ten. A positive value k for the Scale operand of Shift results in a left shift by k digit positions (an effective multiplication by 10^k); a negative value results in a right shift by k digit positions (an effective multiplication by 10^{-k}). Constraint_Error is raised if a loss of significance would result from a left shift. Right shifts always succeed.
- There is a rich collection of functions for converting numeric values between decimal and other representations. All of the other database domain classes, except for Real and Smallint but including database character strings, can be interconverted with decimal representations (subject, of course, to constraints). There is also a function to convert to the type Standard.String, but none to convert from Standard.String. To convert a Standard.String object to decimal, first convert it to SQL_Char_Not_Null.

The reasoning behind this selection of types for interconversion of decimal data is as follows. Conversion between other numeric and character types can be accomplished through Ada explicit type conversions and the Image and Value functions and predefined attributes for the integer types. The predefined functions do not exist for interconversion with decimal data, and must be created. The inclusion of SQL_Int_Not_Null in the set of types for which conversion functions exist and the exclusion of SQL_Smallint_Not_Null and Standard.Integer (and the similar choices with respect to the floating point and character string types) from that set is a consequence of the rules of Ada implicit type conversions (see LRM 4.6(15)). Consider the expression To_SQL_Decimal_Not_Null(1). The literal 1 has type *<universal integer>*. It must be converted, implicitly, to a type for which To_SQL_Decimal_Not_Null is defined. Were there more than one such integer type, the implicit conversion would be ambiguous and could not proceed. It would be necessary to write To_SQL_Decimal_Not_Null(Integer (1)), say. As it is assumed that literal operands are common for these functions, since the direct formation of decimal constants is impossible, the inclusion of only one type from each class (integer, floating point, and character string) makes these expressions easier to write.

The conversion functions are described in the following list. Use of these functions will require type conversions to or from SAME base types, as the rules of Ada program derivation do not produce functions with the appropriate parameter profiles. Sections 3.8 and 5.6.2 describe these type conversions.

- The function To_SQL_Char_Not_Null returns a printable form of a decimal value as an object of the type SQL_Char_Pkg.SQL_Char_Not_Null. The function is modeled after the Image functional attribute and the Float_lo put routines. Leading zeroes to the left of the decimal point are suppressed, unless all such digits are zero, in which case a single zero appears; a leading position is reserved for a sign character which is blank for non-negative values and '-' for negative values; all digits to the right of the decimal point appear for all values; a decimal point does not appear for integers, i. e., for objects with a scale of zero.
- The function To_String is modeled after the To_SQL_Char_Not_Null function, but returns an object of type Standard.String.

- The functions `To_SQL_Double_Precision_Not_Null` and `To_SQL_Int_Not_Null` return objects of types `SQL_Double_Precision_Pkg.SQL_Double_Precision_Not_Null` and `SQL_Int_Pkg.SQL_Int_Not_Null`. Conversion to integer rounds to the nearest integer; it raises `Constraint_Error` if the decimal value is too large in absolute magnitude to be stored as an object of type `SQL_Int_Pkg.SQL_Int_Not_Null`. Conversion to float truncates, but does not raise any exceptions.
- The function `To_SQL_Decimal_Not_Null` taking an operand of type `SQL_Char_Pkg.SQL_Char_Not_Null` requires its operand to be in a special format. The first character must be either a blank, "+" character, a numeric character (i.e., a character in the range "0" .. "9"), a decimal point or period ("."), or the character "-." The last possibility signifies a negative quantity; the remaining possibilities signify a non-negative quantity. (The strings "+0.0" and "-0.0" are acceptable and indicate the value zero.) The remaining characters must all be numeric, with the possible exception of a period. There can be no more than one period anywhere in the string, although there may be none. Violation of any of these restrictions will cause `Constraint_Error` to be raised. The scale of the result is the number of characters appearing after the period, if present. Thus the strings "9." and "9" both have scale zero, whereas "9.0" has scale one. All three strings represent the same quantity. This function is such that `To_SQL_Decimal_Not_Null(To_SQL_Char_Not_Null(x)) = x`, for `x` of the `SQL_Decimal_Not_Null` type.
- The function `To_SQL_Decimal_Not_Null` taking a parameter of type `SQL_Int_Pkg.SQL_Int_Not_Null` always returns an object of scale zero. The equality `To_SQL_Int_Not_Null(To_SQL_Decimal_Not_Null(x)) = x`, where `x` is of type `SQL_Int_Pkg.SQL_Int_Not_Null`, is valid. On the other hand, the equality `To_SQL_Decimal_Not_Null(To_SQL_Int_Not_Null(x)) = x` holds only if `x` has an integral value and `Constraint_Error` is not raised on the conversion to integer.
- The function `To_SQL_Decimal_Not_Null` taking `SQL_Double_Precision_Pkg.SQL_Double_Precision_Not_Null` raises `Constraint_Error` if its input is too large in absolute magnitude to be represented by the `SQL_Decimal_Not_Null` type. The scale for inputs with negative exponents is calculated as the exponent of the input value (in Ada normal form, LRM 14.3.8) minus the quantity `SQL_Double_Precision_Not_Null'Digits - 1`. The scale for results with positive exponents is 0. These conversion functions are inaccurate and the equalities `To_SQL_Decimal_Not_Null(To_SQL_Double_Precision_Not_Null(x)) = x` and `To_SQL_Double_Precision_Not_Null(To_SQL_Decimal_Not_Null(x)) = x` do not in general hold.
- The function `Width` assists in printing decimal values. The equality `Width(x) = To_SQL_Char_Not_Null(x)'Length` is valid.
- The function `Integral_Digits (Scale)` returns the number of digits to the left (right) of the decimal point as defined by the type of the operand. These functions' values depend only on the type, not the value, of their operands. The function `Fore (Aft)` returns the number of significant digits to the left (right) of the decimal point. These functions consider leading (trailing) insignificant zeroes.

Fore returns one if there are no significant digits in the integer portion of the input value. Aft returns one if there are no significant digits in the fractional portion. Thus `Fore(To_Decimal_Not_Null("0.0")) = 1` and `Aft(To_Decimal_Not_Null("0.0")) = 1`.

- The functions `Machine_Rounds` and `Machine_Overflows` mimic the predefined Ada floating point type attributes. They are both the constant function `true` on VAX and IBM machines.

3.5.2. SQL Support

The `SQL_Decimal_Pkg` defines a null bearing type, `SQL_Decimal`, in the usual way. Arithmetic and comparison operators are defined for this type with their usual semantics. Conversion functions are likewise defined. The semantics of the conversion functions are the same as their counterparts defined with respect to `SQL_Decimal_Not_Null` for non-null values. Conversion functions for `SQL_Decimal` exist with respect to all of the non-null bearing types described in the list given above and also their null bearing counterparts. For the conversions from `SQL_Decimal`, these functions are distinguished by name. Thus `To_SQL_Char` as defined in `SQL_Decimal_Pkg` takes an operand of a type derived from `SQL_Decimal` and returns an object of type `SQL_Char_Pkg.SQL_Char`; whereas `To_SQL_Char_Not_Null` returns an object of type `SQL_Char_Pkg.SQL_Char_Not_Null`. Symmetrically, there are overloads of `To_SQL_Decimal` taking `SQL_Char_Pkg.SQL_Char`, `SQL_Char_Pkg.SQL_Char_Not_Null`, `SQL_Int_Pkg.SQL_Int`, and `SQL_Int_Pkg.SQL_Int_Not_Null`, etc. These functions are distinguished by their parameter profiles. For the conversion functions interconverting `SQL_Decimal` with other null bearing types, if the input is the null value, the result is the null value. The functions which convert `SQL_Decimal` object to non-null bearing types raise `Null_Value_Error` on the null input.

An abstract domain based on a BCD concrete representation is constructed from two type definitions, two subtype definitions, and a package instantiation in the standard manner. The types are defined without a discriminant constraint, which is provided by the subtype definitions. The discriminant specifies the scale of the type. Just as SQL character string columns have fixed length, SQL decimal columns have fixed scale. Therefore objects are declared to be of the subtypes.

Example

Suppose the Weight of a part is stored, in decimal, in tenths of some weight unit. The Weight abstract domain is defined by the following set of definitions, assumed to appear in a domain definition package within the scope of a `use` for `SQL_Decimal_Pkg`.

```
Weight_Scale : constant decimal_digits := 1;
type WeightNN_Base is new SQL_Decimal_Not_Null;
subtype Weight_Not_Null is WeightNN_Base (scale => Weight_Scale);
type Weight_Base is new SQL_Decimal;
subtype Weight_Type is Weight_Base (scale => Weight_Scale);
package Weight_Ops is new SQL_Decimal_Ops
  (Weight_Base,
   WeightNN_Base,
   in_scale => Weight_Scale);
```

Notice the use of a constant to define the scale value for the two subtypes. There is no way to define one of those values in terms of the other, as there was for character string based domains. Notice also that the unconstrained types, not the constrained subtype, are passed as the actual type parameter. The generic formal `in_scale` will be described below, as part of the discussion of range constrained assignment.

3.5.3. Range Constraints for Decimal Types

Range constrained assignment is implemented in a novel way for decimal types. This is because the type `SQL_Decimal_Not_Null` is not a visible Ada numeric type, as the other numeric `_Not_Null` types are. Thus, types derived from `SQL_Decimal_Not_Null` cannot be directly constrained. Range constraints for decimal types are provided by parameters passed to the instantiation of the generic `_Ops` package. As can be seen from inspection of the generic specification shown in Figure 3-4, there are seven such parameters. (The procedure parameters should default, as they do for the other generic `_Ops` packages.) The use of these parameters is as follows.

- **`in_scale`**: gives the scale of the high and low values of the range. That scale need not be the same as the scale of the type. However, it is good practice to assign this parameter the scale of the type. For types without explicit range constraints, this is all that need be done.
- **`first_sign`, `first_integral`, `first_fractional`**: gives the sign ("-", "+") of the low value of the range, the (unsigned) value of the integral portion of the low value of the range (the portion to the left of the decimal point) and the value of the fractional portion of the low value of the range, the portion to the right of the decimal point.
- **`last_sign`, `last_integral`, `last_fractional`**: as above, but for the high order value of the range.

The defaults for these parameters are arranged to be the smallest (most negative) and largest values which can be represented in the underlying decimal representation. Thus if no values are given for these parameters, the domain is unconstrained.

The four parameters making up the two unsigned values defining the range are defined as restricted strings (`Numeric_String`). This type allows only character strings containing decimal digits. It is defined in `SQL_Decimal_Pkg` as is the type `Sign_Character`, an enumeration type having the values "-" and "+." The format of the generic parameters was chosen to avoid runtime errors. Were these values passed as two objects of type string, then malformed values could not be detected at compile time.

The actual parameters are converted to decimal format during the elaboration of the instantiated package by the *sequence_of_statements* in the package body. This means that the conversion is done at run time, but only once during program execution. The objects into which they are converted are local.

Example

Suppose that we wished to constrain the `Weight` domain defined earlier to allow only non-negative values. We might then code the package instantiation with


```

package Weight_Ops is new SQL_Decimal_Ops
    (Weight_Base,
     WeightNN_Base,
     in_scale => Weight_Scale,
     first_sign => '+',
     first_integral => "0",
     first_fractional => "0");

```

The remaining parameters may be allowed to default.

There is no check performed that the value defined by the combination `first_sign`, `first_integral`, `first_fractional` is in fact less than or equal to the value defined by `last_sign`, `last_integral`, `last_fractional`. If that relation does not hold, any attempt to use the generated assign procedures will cause a runtime `Constraint_Error`.

Instantiation of the generic `_Ops` package creates membership test functions, `Is_In`, on the types `SQL_Decimal` and `SQL_Decimal_Not_Null`. These functions may be used to prevent assign procedure calls from raising `constraint_error`. Supposing that an object `A_Decimal_Object` has some type derived from `SQL_Decimal`. To ensure that it can be safely assigned to the object `A_Weight`, of type `Weight_type`, one can code

```

if Is_In(Weight_Type(A_Decimal_Object)) then
    assign(A_Weight, Weight_Type(A_Decimal_Object));
end if;

```

The syntax of the Ada membership test is `<object_identifier> in <type_mark>`. As the membership test cannot be overloaded, this syntax cannot be duplicated. The allowed syntax is, however, a close approximation. The test that an object `x` may be safely assigned to an object of type `T` is coded `Is_In(T(x))`, which is self-explanatory.

The `Is_In` function which takes the null bearing type `SQL_Decimal` returns `Boolean`, not `Boolean_With_Unknown`. If the object passed to the function is in fact null, then `Is_In` returns **true**. This is because assignment of the null value to a null bearing object will not raise `constraint_error`.

```

generic
type With_Null_Type(scale : decimal_digits) is limited private;
type Without_Null_Type(scale : decimal_digits) is limited private;
in_scale          : decimal_digits := 0;
first_sign        : Sign_Character := '-';
first_integral    : Numeric_String :=
    (1..decimal_digits'last-in_scale => '9');
first_fractional  : Numeric_String :=
    (1..in_scale => '9');
last_sign         : Sign_Character := '+';
last_integral     : Numeric_String :=
    (1..decimal_digits'last-in_scale => '9');
last_fractional   : Numeric_String :=
    (1..in_scale => '9');
with function Is_In_Base (Right : Without_Null_Type;
    Lower, Upper : SQL_Decimal_Not_Null2)
    return boolean is <>;
with function Is_In_Base (Right : With_Null_Type;
    Lower, Upper : SQL_Decimal_Not_Null2)
    return boolean is <>;
with procedure Assign_with_check
    (Left : in out Without_Null_Type;
    Right : Without_Null_Type;
    Lower, Upper : SQL_Decimal_Not_Null2)
    is <>;
with procedure Assign_with_check
    (Left : in out With_Null_Type;
    Right : With_Null_Type;
    Lower, Upper : SQL_Decimal_Not_Null2)
    is <>;
with function To_SQL_Decimal_Not_Null2 (Value : Without_Null_Type)
    return SQL_Decimal_Not_Null2 is <>;
with function To_SQL_Decimal_Not_Null2 (Value : With_Null_Type)
    return SQL_Decimal_Not_Null2 is <>;
with function To_SQL_Decimal_Not_Null (Value : SQL_Decimal_Not_Null2)
    return Without_Null_Type is <>;
with function To_SQL_Decimal (Value : SQL_Decimal_Not_Null2)
    return With_Null_Type is <>;
package SQL_Decimal_Ops is
    procedure Assign (Left : in out Without_Null_Type;
        Right : Without_Null_Type);
    procedure Assign (Left : in out With_Null_Type;
        Right : With_Null_Type);
    function Is_In (Right : Without_Null_Type)
        return boolean;
    function Is_In (Right : With_Null_Type)
        return boolean;
    function With_Null (Value : Without_Null_Type)
        return With_Null_Type;
    function Without_Null (Value : With_Null_Type)
        return Without_Null_Type;
end SQL_Decimal_Ops;

```

Figure 3-4: The Generic Subpackage SQL_Decimal_Ops

3.6. Data Types Not in the SQL_Standard

The previous sections deal with the data types supported by ANSI standard SQL [2]. Many database management systems extend the standard to other types and some support the standard types, particularly the string type, in non-standard ways. This section outlines the way in which a user of the SAME can extend the data typing facilities. This is done by providing a package which supports the new type.

To design a new support package, one must first decide on the database representation of the type and on the method by which null values of the type will be represented. It is likely that the database representation can be simulated by one of the types in SQL_STANDARD. If this is not possible or desirable, a new package, with the name `DBMS_Standard`,²³ should be constructed to contain the concrete, database representation as an Ada type.

It is strongly recommended that the null value representation be safe, in the sense that null values cannot inadvertently and incorrectly be used as though they were not null. This suggests an abstract, private type to represent domain values at the abstract interface. If that route is chosen, the support package should include null testing functions `Is_Null` and `Not_Null` and conversion functions `With_Null` and `Without_Null`. A null value for the type should also be available in the package specification. In the SAME standard packages discussed so far, the null values `Null_SQL_Int`, `Null_SQL_Char`, etc., are defined as parameterless functions, rather than as private constants. This treatment causes a null value to be created for each type derived from the types in the SAME standard packages. In every case, a function for converting a non-null value from the concrete representation to the abstract one should be provided to the builders or abstract modules.

If the model of the previous sections is followed, i. e., if each abstract domain has two type representatives, a `_Not_Null` visible Ada type and a private `_Type` supporting nulls, generating the conversion functions `With_Null` and `Without_Null` by generic instantiation will tie the two types together. Other functions supplied by the package will depend on the nature of the type being defined and the designer's choice.

3.6.1. Ada Enumeration Types

This section illustrates user extensions to the SAME typing model with an implementation of Ada enumeration types. Enumeration types can be represented in the database as either an integer or as a character string. The integer encoding will save space but will be incomprehensible to any non-Ada database applications. The character string representation will cost space, but will make the type meaningful to other applications, such as any interactive SQL tool or report writer supplied by the database vendor. The representation decision must be made at database design time, so that the proper column definitions can be made. This decision can be made separately for each enumeration type to be stored in the database.

The treatment chosen for the null value parallels the treatment in the standard packages. A limited private record type definition encapsulates the enumeration type with a Boolean. As the type is private, the enumeration value can be accessed only through the functions provided.

²³e.g., `Ingres_Standard`, `Oracle_Standard`, `DB2_Standard`, etc.

The treatment uses the enumeration type itself as the `_Not_Null` type. It defines both the three-valued (`Boolean_with_Unknown`) and the two-valued (`Boolean`) comparison operators (`Equals`, `Not_Equals`, (or `=`, `/=` (implicitly)) `<`, `<=`, etc), and the functions `Succ`, `Pred`, `Pos`, `Val`, `Image` and `Value` for the limited private `_Type`. These last two functions (`Image` and `Value`) are also defined for the `_Not_Null` type. These functions take (`Value`) and return (`Image`) objects of the SAME predefined types `SQL_Char` (or `SQL_Char_Not_Null` when applied to the `_Not_Null` type). This usage is to accommodate character set independent programs.

The specification for the package `SQL_Enumeration_Pkg` appears in Figure 3-5. It is a generic package with the enumeration type as the formal parameter. Even if the limited private type were declared with no operations other than the test and conversion functions, it would still be necessary to make this package a generic. The body of the package appears in Appendix C.

Example

Suppose the Status of a supplier has only a small number of legal values. This can occur even if the database design was not developed with Ada in mind. It may be known to application developers that a Status of zero indicates an unacceptable supplier, five an acceptable supplier and ten a preferred supplier. This information will be hidden in the application code. Ada allows this knowledge to be made visible in the type definition while freeing the application programmer from the need to know it. The Status abstract domain may be encoded as follows.

```
type Status_Not_Null is (Unacceptable, Acceptable, Preferred);
for Status_Not_Null use
    (Unacceptable => 0,
     Acceptable   => 5,
     Preferred    =>10);
package Status_Pkg is new SQL_Enumeration_Pkg(Status_Not_Null);
type Status_Type is new Status_Pkg.SQL_Enumeration;
```

Notice that the `_Type` is derived from the private type generated from the package instantiation. This gives the two types making up the abstract domain similar, conventional names. It also means that the package instantiation need not be made visible to the application program (see Chapter 5).

The task of converting from the database representation, in this case `SQL_Standard.Int` (or possibly `SQL_Standard.Smallint`), to the abstract representation, the types `Status_Not_Null` or `Status_Type`, is the responsibility of the abstract module. Section 4.2 describes these modules. In this case, the integer representation to be used on the database is that given by the `for ... use` representation clause. It is necessary to use `Unchecked_Conversion` to accomplish this.²⁴ `Unchecked_Conversion` is a predefined generic function. Its use is illustrated in the following template.

²⁴`Unchecked_Conversion` is a Chapter 13 feature. Care must be taken in its use.

```

with Unchecked_Conversion;
...
function Cnvrt_Status_In is new
    Unchecked_Conversion (Integer, Status_not_Null);
function Cnvrt_Status_Out is new
    Unchecked_Conversion (Status_not_Null, Integer);
...
begin
...
    <Application Variable> :=
        With_Null(Cnvrt_Status_In(<Database Variable>));
...
    <Database Variable> :=
        Cnvrt_Status_Out(Without_Null(<Application Variable>));
...
end;

```

These assignment statements assume that the database value involved is not null. See Section 4.2 for more details.

It is possible to use the position (POS) of an enumeration literal within the enumeration type instead of its representation as the database encoding, if the database is being defined with the Ada applications. Use of the representation encoding may help prevent inadvertent changes in the enumeration type definition from destroying the meaning of the database.

If the character string representation is chosen, the mapping between database and internal representations is accomplished with the Image and Value functions created by the instantiation of SQL_Enumeration_Pkg. Care must be taken to ensure the database columns storing these strings are long enough to accommodate growth. Care must also be taken to strip or pad blanks as needed and to ensure the case of the database string is such that non-Ada programs, which may be case sensitive, can recognize them. Although character string representation takes more space, it has the advantage of being readable by non Ada programs and is relatively impervious to changes in the enumeration type, provided enough space has been reserved initially.

3.6.2. Date Time Types

Many database management systems extend the ANSI standard by offering a date - time data type. The follow-on standard, SQL2, under development by ANSI [4], also provides a date - time data type. This section develops support for date - time types as yet another example of user extensions to the SAME. As no standard treatment of date - time has been established, two distinct support packages are presented here. One of the packages supports the SQL2 date - time data type; the other supports Ingres date - time.

The two support packages have a lot in common. In both cases, values appear at the concrete interface as character strings. Therefore, in both cases, the concrete type used to store dates is a derived type of SQL_Char_Not_Null. In both cases, limited private types are declared which support

- **Null values for date - times.** The test and conversion functions and three-valued logic and arithmetic are supported (see Section 3.1).
- **Date time arithmetic.** The DBMS date time arithmetic is defined by appropriate functions and operators.

```

with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
generic
  type SQL_Enumeration_Not_Null is (<>);
package SQL_Enumeration_Pkg
  is
    ---- Possibly Null Enumeration ----
    type SQL_Enumeration is private;
    function Null_SQL_Enumeration return SQL_Enumeration;

    -- conversion functions
    function Without_Null(Value : in SQL_Enumeration)
      return SQL_Enumeration_Not_Null;
    -- raises Null_Value_Error on the null input
    function With_Null(Value : in SQL_Enumeration_Not_Null)
      return SQL_Enumeration;

    procedure Assign (Left : in out SQL_Enumeration;
                     Right : in SQL_Enumeration);

    -- Three-valued comparison operators; raise no exceptions
    function Equals (Left, Right : SQL_Enumeration)
      return Boolean_with_Unknown;
    function Not_Equals (Left, Right : SQL_Enumeration)
      return Boolean_with_Unknown;
    function "<" (Left, Right : SQL_Enumeration)
      return Boolean_with_Unknown;
    function ">" (Left, Right : SQL_Enumeration)
      return Boolean_with_Unknown;
    function "<=" (Left, Right : SQL_Enumeration)
      return Boolean_with_Unknown;
    function ">=" (Left, Right : SQL_Enumeration)
      return Boolean_with_Unknown;

    function Is_Null (Value : SQL_Enumeration) return Boolean;
    function Not_Null (Value : SQL_Enumeration) return Boolean;

    -- 'Pred, 'Succ return the null value on the null input
    -- 'Image, 'Pos raise Null_Value_Error on the null input
    function Pred (Value : in SQL_Enumeration)
      return SQL_Enumeration;
    function Succ (Value : in SQL_Enumeration)
      return SQL_Enumeration;
    function Pos (Value : in SQL_Enumeration)
      return Integer;
    function Image (Value : in SQL_Enumeration)
      return String;
    function Val (Value : in Integer)
      return SQL_Enumeration;
    function Value (Value : in String)
      return SQL_Enumeration;
  private
    type SQL_Enumeration is record
      Is_Null: Boolean := true;
      Value: SQL_Enumeration_Not_Null;
    end record;
end SQL_Enumeration_Pkg;

```

Figure 3-5: The Package Specification SQL_Enumeration_Pkg

The definitions of the limited private types are optimized for doing arithmetic. The visible, `_Not_Null` types, derived from `SQL_Char_Not_Null`, are optimized for displays. Both packages contain `_Ops` generic subpackages for generating conversion functions between the `_Not_Null` and `_Type` types. Both packages also contain functions for converting be-

tween the `_Type` and the most nearly appropriate predefined Ada types, `Calendar.time` and `Standard.duration`. These conversions are necessarily inexact.

Support for the SQL2 date - time type is provided by the package `SQL_Date_Pkg`, the specification of which can be found in Appendix C. SQL2 defines two date - time types, `Date` and `Interval`. A date is a specific moment in time; an interval is a period of time. Both of these types can be modified by a so-called "date-time qualifier." This qualifier specifies the precision of a date or interval. Date-time qualifiers specify the most and least significant portions of a date or interval to be recorded. A database table column having date or interval type has an associated date time qualifier. Thus, all values in the column have the same format. See [4] for more details.

The declaration of an abstract domain for date or interval types must also include date-time qualifier information. The discriminants of the types `SQL_Date` and `SQL_Interval` capture that information. The discriminants are specified in the associated type declarations within the abstract domain declaration, as exemplified by the following domain package.

```
with SQL_Date_Pkg; use SQL_Date_Pkg;
package Date_Domain is

    type DateNN_Base is new SQL_Date_Not_Null;
    subtype Date_Not_Null is DateNN_Base (1..10);
    type Date_Type is new SQL_Date (From=>year, To=>Day,
                                   Fractional=>0);
    package Date_Ops is new SQL_Date_Ops (Date_Type, DateNN_Base);

    type MonthsNN_Base is new SQL_Date_Not_Null;
    subtype Months_Not_Null is MonthsNN_Base(1..2);
    type Months_Type is new SQL_Interval(From=>Month, leading=>2,
                                         To=>Month, Fractional=>0);
    package Months_Ops is new SQL_Date_Ops
        (Months_Type, MonthsNN_Base);

    package Date_Months_Ops is new
        SQL_Date_Interval_Ops (Date_Type, Months_Type);

end Date_Domain;
```

Here objects of `Date_Type` record a year, a month, and a day. The `_Not_Null` string version of `Date` is ten characters long, as SQL2 defines the character representation of such dates to have the form `yyyy-mm-dd`. Objects of `Months_Type` are intervals recorded in months. Intervals from 0 to 99 months can be recorded as objects of `Months_Type`.

As before, the generic subpackage `SQL_Date_Ops` generates conversion functions between the `_Not_Null` and `_Type` types of a domain. The generic subpackage `SQL_Date_Interval_Ops` generates arithmetic functions on the date and interval types which are the actual type parameters. In order for the application program to do date arithmetic such as adding or subtracting an interval to or from a date and subtracting two dates to form an interval, an instantiation of `SQL_Date_Interval_Ops` for the types must exist in the domain package. This "cross product" will not require very many package instantiations, as there are likely to be very few distinct date or interval domains. Most dates and intervals are inherently comparable.

The following example shows how the Date_Domain can be used.

```
with Date_Domain; use Date_Domain;
with text_io; use text_io;
procedure use_dates is

    use Date_Ops, Months_Ops, Date_Months_Ops;

    Today_Not_Null : Date_Not_Null := to_sql_char_not_null("1988-10-25");;
    Today : Date_Type;
    Two_Months_Not_Null : Months_Not_Null := to_sql_char_not_null(" 2");
    Two_Months : Months_Type;

begin
    Parse_And_Assign(Two_Months, Two_Months_Not_Null);
    Parse_And_Assign(Today, Today_Not_Null);
    put_line(to_string(without_Null(Today + Two_Months)));;
end use_dates;
```

Notice that, as a derived types of SQL_Char_Not_Null, Date_Not_Null and Months_Not_Null inherit conversion functions from and to the predefined type string. The procedure Parse_And_Assign replaces the functions With_Null in other support packages. This procedure uses the discriminants of the left, output operand to determine the meaning of the right, character string input operand. Parse_And_Assign can raise Constraint_Error if the output discriminants are not legal according to the rules of SQL2.

The Ingres date time data type is supported by a package Ingres_Date_Pkg, the specification of which can be found in Appendix C. Ingres dates are markedly different from SQL2 dates. There is only one type, rather than two, and row columns of date type may contain either dates or intervals. Further, the dates and intervals have varying formats. Thus, to determine the meaning of a given value of a date column, it is necessary to examine the value. See [13] for details.

Ingres_Date_Pkg defines a single limited private type, Ingres_Date, for holding values of Ingres date columns. As earlier, this type is optimized for date arithmetic; whereas Ingres_Date_Not_Null is optimized for display. The discriminant of the type Ingres_Date is used to record the nature of a value in an object of the type. The type of the discriminant, Ingres_Date_Format, is an enumeration type having the value set (Datetime, Interval, Unknown). The Ingres_Date type definition specifies a default of Unknown for the discriminant. Variables of Ingres_Date type can be declared without discriminant constraints. Such variables can contain either dates or intervals, just as Ingres database columns of type date can contain either class of values. The declaration of an abstract domain based on an Ingres date type is illustrated by the following.

```
with Ingres_Date_Pkg; use Ingres_Date_Pkg;
package Ingres_Date_Domain is

    type Date_Not_Null is new Ingres_Date_Not_Null;
    type Date_Type is new Ingres_Date;
    package Date_Ops is new Ingres_Date_Ops (Date_Type, Date_Not_Null);

end Ingres_Date_Domain;
```

Notice that the _Not_Null type is already constrained by the definition of Ingres_Date_Not_Null. All Ingres dates and intervals are exactly 25 characters in length. There is no need for a cross product package as there was for SQL2. The following program uses Ingres dates.


```

with Ingres_Date_Domain; USE Ingres_Date_Domain;
with Text_IO; USE Text_IO;
procedure Use_Ingres_Dates is

  use Date_Ops;

  Date_String1 : string(Date_Not_Null'Range) := "1988-oct-25" &
    (12..Date_Not_Null'Last => ' ');
  Date_String2 : string (Date_Not_Null'Range) := "2 mm" &
    (5..Date_Not_Null'Last => ' ');
  Date1_Not_Null : Date_Not_Null := to_sql_char_not_null(Date_String1);
  Date2_Not_Null : Date_Not_Null := to_sql_char_not_null(Date_String2);
  Date1, Date2 : Date_Type;

begin
  assign(Date1, With_Null(Date1_Not_Null));
  assign(Date2, With_Null(Date2_Not_Null));
  put_line(to_string(without_null(Date2 + Date1)));

end Use_Ingres_Dates;

```

Both treatments of the date - time type presented in this section have as their design goal the creation of an abstract type which simulates a database type. Thus the types and operations in SQL_Date_Pkg simulate SQL2's treatment of dates; the types and operations in Ingres_Date_Pkg simulate Ingres' treatment of dates. Applications using these packages can operate on dates in the same way that the DBMS does.

In constructing new data type support packages, the user of the SAME is free to substitute *other design goals for that of DBMS simulation*. For example, it may be desirable to construct a type support package for use with Ingres that makes its date type more closely resemble the emerging SQL2 standard. Such a support package may improve the portability of applications which use it. (Of course, it will not make the Ingres SQL portion of the application treat dates in the style of SQL2.) The user is permitted to extend the SAME with non-standard data types in any way that he or she sees fit. It is strongly suggested that such extensions maintain the safe treatment of nulls which is a defining characteristic of the SAME standard packages.

3.7. Packaging the Type Definitions

Prior sections deal with data definition at the level of the individual abstract domains. This section begins the process of describing the database at higher level of granularity. The level of the tuple or row is not described until Chapter 4; the level of the relation or table is never reached, as Ada programs do not deal with tables as a whole, but only with rows within tables, one at a time.

The identification of the abstract domains over which a database is defined occurs during the database design process. Most database design methodologies lose this information however, as database technology has evolved without regard to the needs of strongly typed languages such as Ada. In developing the Ada description of the database for use with the SAME, it may be necessary to retro-fit this information. This section assumes that the Ada description is developed from the SQL description.

The first problem to be addressed is the re-identification of the abstract domains. In the example developed in the introduction (see Figure 1-6), the abstract domains are identified by the attribute or column names. Thus the columns named PNO in the tables P and SP have the same abstract domain; so do the columns named CITY in the tables S and P. Reliance on column names is not recommended. There is no rule in database design methods nor in SQL that enforces or even suggests such column-naming practices. In general, the problem is determining whether any given pair of columns share an abstract domain.

The number of columns in a real world database description is generally quite large and the task of examining each pair is overwhelming. Most such pairs are obviously not over the same domain, making the task simpler than this crude analysis suggests. There is one case in which columns from two distinct table definitions are obviously over the same domain: the foreign key. A foreign key is a column of one table, the values of which are keys of another table. These columns clearly have the same domain. In the example, SNO and PNO are foreign keys in the SP table. It is for this reason that the PNO columns of P and SP have the same domain.

Once the foreign keys are recognized, remaining column pairs must be decided on a case-by-case basis. The rule to follow is the comparison rule: "Does it make sense to compare values of these columns?" If the answer is yes, the columns probably have the same domain. For this reason, the columns CITY in S and P of the example can be seen to have the same domain. This rule frequently applies to fields containing dates. The Date_Created and Date_Modified columns of a record describing a product are probably over the same domain. On the other hand, the Birth_Date column of an employee record may well have a different domain. It is the designer's responsibility to make these determinations.

Once the abstract domains have been identified and the Ada type definitions have been written, the definitions are assembled into packages, called *domain packages*, and compiled into Ada libraries for the use of programmers. The essential rule of these packages is that they must be disjoint; that is, *no abstract domain should be declared in more than one domain package.*²⁵ The reason for this rule is obvious. If the type and package declarations making up an abstract domain declaration are duplicated in more than one package, the result is the declaration of two distinct domains.

There are no hard and fast rules for determining which abstract domain declarations to collect into domain packages. The rule which places each domain declaration into its own package satisfies the disjointness rule, but may result in excessively many packages.

A useful technique is to begin by collecting abstract domains into possibly overlapping sets and then reducing the sets by intersection until a disjoint collection is obtained. The initial collection can be created by letting each base table definition create a set in the collection. An alternative has each set in the original collection correspond to an application view, that is, be the collection of abstract domains of interest to a given application. This alternative requires that the designer have knowledge of the applications to be run against the database. Such information is often available during the database design. The advantage of using application views is that they map naturally to the application programs.

²⁵The declaration of an abstract domain is the declaration of the two types, and for character string data two subtypes, plus the package instantiation, as described in the preceding subsections.

Example

In the Parts Suppliers example, assume the existence of three application views.

1. A Parts view, concerned only with information about Parts.
2. A Suppliers view, concerned only with information about Suppliers.
3. An Orders view, concerned with all the information in the Database.

From these views, the initial collection of sets of domains is as follows.

1. For Parts, the set containing the domains PNO, PNAME, COLOR, WEIGHT and CITY.
2. For Suppliers, the set containing the domains SNO, SNAME, STATUS, and CITY.
3. For Orders, the set containing the domains PNO, SNO, PNAME, SNAME, STATUS, COLOR, WEIGHT, CITY and QTY.

To complete the design of the domain packages, take intersections of these sets. The final design appears in Figures 3-6 and 3-7. The Parts application will bring into context (with) the packages CITY_Definition_Pkg and Parts_Definition_Pkg. The Supplier application will need CITY_Definition_Pkg and Suppliers_Definition_Pkg. The Orders application will need all four packages.

The pattern of Figures 3-6 and 3-7 is common. A few domains will be shared by multiple views. These domains will appear in small packages. The remaining domains will be unique to an application. In most real world relational databases, the majority of the domains are unique to an application.

An application may need domains defined specifically for it. If an application deals only with preferred suppliers, that is, suppliers with Status > 100, the abstract sub-domain Preferred_Status, illustrated in Section 3.3, is such an application-specific domain. Other application-specific domains may arise from SQL expressions (see Section 4.1.1). For the sake of exposition, suppose the Parts table were to contain Length, Width and Height columns and that these columns had the abstract domain Meters. If part volume, (Length*Width*Height), is returned from an SQL statement, its abstract domain is Cubic_Meters. There may be no database column with this domain. The definitions of such application-specific domains can either be included in the package of application-unique database domain definitions or put into a package by themselves.

Except for the rule that states that domain packages must be disjoint, the other rules for the formation of domain packages are heuristics. The smaller the domain packages, the more packages need to be defined and controlled in configuration management. Larger domain packages may cause unnecessary recompilations. In the Parts-Suppliers example, a given program or component of the Parts application may need visibility to WEIGHT but not to COLOR, for example. If, during database evolution, the definition of the COLOR domain is changed, that program or component may be unnecessarily marked for recompilation.

City Abstract Domain

```
with SQL_Char_Pkg; use SQL_Char_Pkg;
package CITY_Definition_Pkg is
  type CITYNN_Base is new SQL_Char_Not_Null;
  subtype CITY_Not_Null is CITYNN_Base (1..15);
  type CITY_Base is new SQL_Char;
  subtype CITY_Type is CITY_Base (CITY_Not_Null'Length);
  package CITY_Ops is new
    SQL_Char_Ops(CITY_Base, CITYNN_Base);
end CITY_Definition_Pkg;
```

QTY Abstract Domain

```
with SQL_Int_Pkg; use SQL_Int_Pkg;
package QTY_Definition_Pkg is
  type QTY_Not_Null is new SQL_Int_Not_Null
    range 0 .. SQL_Int_Not_Null'LAST;
  type QTY_Type is new SQL_Int;
  package QTY_Ops is new
    SQL_Int_Ops(QTY_Type, QTY_Not_Null);
end QTY_Definition_Pkg;
```

Domains Unique to Parts

```
with SQL_Char_Pkg; use SQL_Char_Pkg;
with SQL_Int_Pkg; use SQL_Int_Pkg;
package Parts_Definition_Pkg is

  type PNONN_Base is new SQL_Char_Not_Null;
  subtype PNO_Not_Null is PNONN_Base (1..5);
  type PNO_Base is new SQL_Char;
  subtype PNC_Type is PNO_Base (PNO_Not_Null'Length);
  package PNO_Ops is new
    SQL_Char_Ops(PNO_Base, PNONN_Base);

  type PNAMENN_Base is new SQL_Char_Not_Null;
  subtype PNAME_Not_Null is PNAMENN_Base (1..20);
  type PNAME_Base is new SQL_Char;
  subtype PNAME_Type is PNAME_Base (PNAME_Not_Null'Length);
  package PNAME_Ops is new
    SQL_Char_Ops(PNAME_Base, PNAMENN_Base);

  type COLORNN_Base is new SQL_Char_Not_Null;
  subtype COLOR_Not_Null is COLORNN_Base (1..6);
  type COLOR_Base is new SQL_Char;
  subtype COLOR_Type is COLOR_Base (COLOR_Not_Null'Length);
  package COLOR_Ops is new
    SQL_Char_Ops(COLOR_Base, COLORNN_Base);

  type Weight_Not_Null is new SQL_Int_Not_Null
    range 0 .. SQL_Int_Not_Null;
  type Weight_Type is new SQL_Int;
  package Weight_Ops is new
    SQL_Int_Ops(Weight_Type, Weight_Not_Null);
end Parts_Definition_Pkg;
```

Figure 3-6: The Domain Packages for Suppliers-Parts

Domains Unique to Suppliers

```
with SQL_Char_Pkg; use SQL_Char_Pkg;
with SQL_Int_Pkg; use SQL_Int_Pkg;

package Suppliers_Definition_Pkg is

    type SNONN_Base is new SQL_Char_Not_Null;
    subtype SNO_Not_Null is SNONN_Base (1..5);
    type SNO_Base is new SQL_Char;
    subtype SNO_Type is SNO_Base (SNO_Not_Null'Length);
    package SNO_Ops is new
        SQL_Char_Ops(SNO_Base, SNONN_Base);

    type SNAMENN_Base is new SQL_Char_Not_Null;
    subtype SNAME_Not_Null is SNAMENN_Base (1..20);
    type SNAME_Base is new SQL_Char;
    subtype SNAME_Type is SNAME_Base (SNAME_Not_Null'Length);
    package SNAME_Ops is new
        SQL_Char_Ops(SNAME_Base, SNAMENN_Base);

    type Status_Not_Null is new SQL_Int_Not_Null
        range 0 .. 100;
    type Status_Type is new SQL_Int;
    package Status_Ops is new
        SQL_Int_Ops(Status_Type, Status_Not_Null);
end Suppliers_Definition_Pkg;
```

Figure 3-7: The Domain Packages for Suppliers-Parts, cont'd.

3.8. The Package SQL_Base_Types_Pkg

The method of abstract domains for database description presented in this section will generally produce a large number of distinct abstract types. This is in keeping with good Ada design practice, in which the type of an object gives some indication as to the semantics of its values. Due to Ada's implementation of strong typing, in particular, Ada's lack of polymorphism, this proliferation of types can result in cumbersome programming requirements. There are parts of many applications in which abstract and strong typing are hindrances. These are the parts of the application which lie at low levels of abstraction. Examples are communication protocols and display handlers. These services treat their operands as bit streams or character strings, not as Weights or Names or Part Numbers. It is possible, and may be desirable, to build abstract interfaces to these services for the application. Indeed, the SAME builds just such abstract interfaces for database services. These interfaces are the subject of the next section. Whether abstract interfaces taking operands of abstract types are desirable for other services is a matter for the application designer to decide. It should be noted, however, that such interfaces merely postpone the problem, moving it from the realm of the application to the realm of the implementation of the interface. This can itself be considered an advantage; it is considered an advantage of the SAME.

There are uses, other than the operands of low-level interfaces to low-level services, for operands of concrete types. The result of an SQL COUNT function, for example, often has no obvious abstract type. Such values are inherently comparable; it makes perfect sense to ask whether there are more suppliers in Pittsburgh than there are red parts weighing more than one ton. (It may not be a very interesting question, but it is well defined). It makes no

sense to ask whether "Acme's" supplier number is greater than the part number of "Widgets." Part numbers and supplier numbers are incomparable.

Highly generalized applications are similar to very low-level applications in that they are unconcerned with the specific semantics of the data they manipulate. The classic examples of such generalized applications are *ad hoc* browsing programs. Such programs can be written to be independent of the database schema; hence, they are necessarily independent of the database semantics. Applications such as these are discussed in Chapter 9.

There is yet another need for concrete types in application programs. Certain of the functions described in previous subsections, the Image and Value functions of integer types and the conversion functions for decimal types, have operands defined in the base packages. The application may need visibility to the base type for an Ada explicit type conversion.

These problems could be solved by making the base and concrete type packages, e.g., SQL_Standard, SQL_Int_Pkg, etc., visible to the application program. However, this results in inconsistencies in the set of functions of available to the applications. The types defined in SQL_Standard are not parts of any abstract domain. Only the Ada predefined operators exist for them. The types defined in a base type support package have sets of subprograms defined for them which are slightly different from those in an abstract domain package; the differences are the subprograms generated by the package instantiation that is part of an abstract domain definition. Furthermore, the naming conventions for these types is slightly different from the naming conventions for abstract domain types. To insure consistency in accessing database values, application programs must view all database values through some abstract domain. What is needed is an abstract domain package which creates concrete domains. The package SQL_Base_Types_Package is designed to meet this need. It appears in Figure 3-8.

Notice that the character and decimal domains in Figure 3-8 do not contain constrained subtypes. Abstract domains which define database columns are constrained, since SQL character strings are fixed length and decimal values have fixed scale, given by the SQL column definition. Objects of the types in SQL_Base_Types_Pkg are less specific and more generalized or concrete. Thus, these objects may have any length or scale.

The subtype declarations which do appear in Figure 3-8 serve a different function. They are defined to be the same types as are defined in the base packages. No operations are defined within SQL_Base_Types_Pkg for these subtypes; therefore, applications with visibility to SQL_Base_Types_Pkg do not have visibility to the base operations, but only to the operations for the types defined in that package. The subtypes can be used as the typemarks in an Ada explicit type conversion. The type of the operand of those conversions must be derived from the same base type. Section 5.6.2 illustrates the use of those type conversions.

```

with SQL_Char_Pkg, SQL_Int_Pkg, SQL_Smallint_Pkg, SQL_Real_Pkg,
     SQL_Double_Precision_Pkg, SQL_Decimal_Pkg, SQL_Standard;
package SQL_Base_Types_Pkg is

    package Character_Set renames SQL_Standard.Character_Set;

    type SQL_Int_Not_Null is new SQL_Int_Pkg.SQL_Int_Not_Null;
    type SQL_Int_Type is new SQL_Int_Pkg.SQL_Int;
    package SQL_Int_Ops is new SQL_Int_Pkg.SQL_Int_Ops(
        SQL_Int_Type, SQL_Int_Not_Null);
    subtype SQL_Int_Subtype is SQL_Int_Pkg.SQL_Int;
    subtype SQL_Int_Not_Null_Subtype is SQL_Int_Pkg.SQL_Int_Not_Null;

    type SQL_Smallint_Not_Null is new SQL_Smallint_Pkg.SQL_Smallint_Not_Null;
    type SQL_Smallint_Type is new SQL_Smallint_Pkg.SQL_Smallint;
    package SQL_Smallint_Ops is new SQL_Smallint_Pkg.SQL_Smallint_Ops(
        SQL_Smallint_Type, SQL_Smallint_Not_Null);
    subtype SQL_Smallint_Subtype is SQL_Smallint_Pkg.SQL_Smallint;
    subtype SQL_Smallint_Not_Null_Subtype is
        SQL_Smallint_Pkg.SQL_Smallint_Not_Null;

    type SQL_Real_Not_Null is new SQL_Real_Pkg.SQL_Real_Not_Null;
    type SQL_Real_Type is new SQL_Real_Pkg.SQL_Real;
    package SQL_Real_Ops is new SQL_Real_Pkg.SQL_Real_Ops(
        SQL_Real_Type, SQL_Real_Not_Null);
    subtype SQL_Real_Subtype is SQL_Real_Pkg.SQL_Real;
    subtype SQL_Real_Not_Null_Subtype is SQL_Real_Pkg.SQL_Real_Not_Null;

    type SQL_Double_Precision_Not_Null is
        new SQL_Double_Precision_Pkg.SQL_Double_Precision_Not_Null;
    type SQL_Double_Precision_Type is
        new SQL_Double_Precision_Pkg.SQL_Double_Precision;
    package SQL_Double_Precision_Ops is
        new SQL_Double_Precision_Pkg.SQL_Double_Precision_Ops(
            SQL_Double_Precision_Type,
            SQL_Double_Precision_Not_Null);
    subtype SQL_Double_Precision_Subtype is
        SQL_Double_Precision_Pkg.SQL_Double_Precision;
    subtype SQL_Double_Precision_Not_Null_Subtype is
        SQL_Double_Precision_Pkg.SQL_Double_Precision_Not_Null;

    type SQL_Char_Not_Null is new SQL_Char_Pkg.SQL_Char_Not_Null;
    type SQL_Char_Type is new SQL_Char_Pkg.SQL_Char;
    package SQL_Char_Ops is new SQL_Char_Pkg.SQL_Char_Ops(
        SQL_Char_Type, SQL_Char_Not_Null);
    subtype SQL_Char_Subtype is SQL_Char_Pkg.SQL_Char;
    subtype SQL_Char_Not_Null_Subtype is SQL_Char_Pkg.SQL_Char_Not_Null;

    type SQL_Decimal_Not_Null is new SQL_Decimal_Pkg.SQL_Decimal_Not_Null;
    type SQL_Decimal_Type is new SQL_Decimal_Pkg.SQL_Decimal;
    package SQL_Decimal_Ops is new SQL_Decimal_Pkg.SQL_Decimal_Ops(
        SQL_Decimal_Type, SQL_Decimal_Not_Null);
    subtype SQL_Decimal_Subtype is SQL_Decimal_Pkg.SQL_Decimal;
    subtype SQL_Decimal_Not_Null_Subtype is
        SQL_Decimal_Pkg.SQL_Decimal_Not_Null;

end SQL_Base_Types_Pkg;

```

Figure 3-8: The Package SQL_Base_Types_Pkg

4. The SAME Operational Model

The previous sections specify the data definition process within the SAME. That process results in a description of the database contents in Ada terms, thereby allowing the Ada programmer to manipulate database data under the control of Ada's strong typing paradigm. The Ada descriptions do not require any conversions of data representation and the treatment of incomplete information prevents any use of null values as though they were not null.

This chapter describes the construction of abstract interfaces and abstract modules. Whereas the data definitions are used by all applications, an abstract interface and its implementation, an abstract module, are specific to a given set of applications.

Applications implemented using the SAME divide the problem into two parts: the part to be solved in Ada and the part to be solved in SQL. The SQL portion of the solution is a collection of procedures the bodies of which are individual SQL statements. This collection is called a *module* in ANSI standard SQL [2]. In the SAME, it is called a *concrete module*, to distinguish it from the abstract module which the Ada programmer sees.

4.1. Constructing an Abstract Interface

For expository purposes it suffices to think of an abstract interface as a package specification and an abstract module as a package body. In practice it is frequently advantageous to construct an abstract interface as a collection of packages. The concrete interface is the Ada package specification of the SQL concrete module. It should be noted that the ANSI standard requires that there be only one concrete module in any application program ([2] Section 4.8).

The abstract interface contains two kinds of declarations: declarations of row record types and declarations of procedures. The procedure declarations of the abstract interface are one for one with the procedures of the concrete module. For each SQL statement in the concrete module there is a procedure declaration in the abstract interface and, in the body, a call to that SQL statement.

A higher level, more abstract and application-oriented interface than that of the abstract interface is conceivable. The application designer may very well wish to create such an additional layer that defines such an interface for his application. The SAME abstract interface does not attempt to "improve" SQL. An abstract module should deal only with the details of database interaction and should never contain application logic.

A procedure declared in the abstract interface has a parameter profile which differs from that of the procedure in the concrete interface that it calls. Parameters declared in the concrete interface have types defined in the package `SQL_Standard` (see Figure 2-2, [5], [16]). The types of parameters and parameter components of procedures declared in the abstract interface are the abstract types described in the previous sections of these guidelines. Beyond that change are two other significant differences in the parameter profiles of procedures at the concrete and abstract interfaces.

1. At the abstract interface, rows being returned from the database or inserted into it are transmitted as record objects rather than individual fields. These

records are called *row records* and their types are the row record types declared in the abstract interface. Every component of the record type must have its value set, either in the abstract module or in the application program, as appropriate. In the case of data being transmitted from the database to the program, i.e., from an SQL `FETCH` or `SELECT` statement, the components of the row record type are one for one with the elements of the *<target list>* of the statement. Similar comments apply to the `INSERT ... VALUES` SQL statement.

2. The `SQLCODE` parameter does not appear at the abstract interface. An optional result parameter appears instead. A full description of this parameter can be found in Section 4.3.

For concreteness, Figure 4-1 lists each executable statement of ANSI Standard SQL [2] and gives the parameters such statements take as abstract procedures along with the parameter modes. Parameters listed as having mode `In out` are logically `out` parameters of a limited type. (They are row records whose components will be of limited types.) Each such procedure may also take, in addition to those listed, a result parameter as the last parameter. The result parameter's mode is always `out`. The phrase *Individual Parameters* indicates that the sequence of individual parameters in the concrete SQL module interface appears as a sequence in the abstract interface, albeit with different types. This treatment is used primarily for runtime parameters of SQL `where` and `having` clauses. Notice that only the *select* statement may take both a row record (for the retrieved row) and a sequence of individual parameters (for the `where` or `having` clause).

SQL Statement	Ada Parameter Kinds	Mode
close	none	
commit	none	
delete - positioned	none	
delete - searched	Individual Parameters	In
fetch	row record	In out
insert values	row record	In
insert (subquery)	Individual Parameters	In
open	Individual Parameters	In
rollback	none	
select	row record	In out
	Individual Parameters	In
update - positioned	Individual Parameters	In
update - searched	Individual Parameters	In

Figure 4-1: Parameter Kinds (with Modes)

4.1.1. A Note on Typing Parameters

It should not generally be difficult to determine the types of the individual parameters and row record components at the abstract interface. If the values of that parameter or component are in transit between the application program and a database column or are compared to a database column in a `where` or `having` clause, the type to be used is one of the abstract types describing the abstract domain underlying that column. If the null value is permissible in the given context, a type supporting null values must be used.

In the case that the value involved is the result of an expression in the SQL statement, particularly one involving more than one database column, the appropriate abstract type may not be obvious. It may be necessary and desirable to create a new type for such an expression (see Section 3.7). The class of that abstract type, e.g. `INT REAL`, etc., can be established from the concrete type of the parameter that holds values of the expression at the concrete interface. The general problem of typing parameters whose values are set by SQL expressions is an instance of the "dimensional analysis" problem. The SAME does not provide its own solution to that problem.

Example

Consider the following problem: "Calculate the total weight of all orders for a given part number." The SQL module specification for this query is:

```
MODULE Concrete__Mod
LANGUAGE Ada

Procedure Calculate_Weight
  PNUMBER Char(5)
  Total_Weight Int
  TW_Indic Smallint
  SQLCODE;

select sum(QTY * WEIGHT)
into Total_Weight INDICATOR TW_Indic
from P, SP
where P.PNO = SP.PNO
and P.PNO = PNUMBER;
```

The concrete interface, that is, the Ada specification of that SQL module is:

```
with SQL_Standard; use SQL_Standard;
package Concrete_Mod is

  procedure Calculate_Weight (PNUMBER : Char;
                             Total_Weight : out Int;
                             TW_Indic out Indicator_Type;
                             SQLCODE : out SQLCODE_Type);

end Concrete_Mod;
```

The abstract interface for this procedure (without the package declaration and context clauses and assuming no result parameter) is:

```
type Weight_Record is record
  Total : Weight_Type;
end record;

procedure Calculate_Weight (PNUMBER : in PNO_Not_Null;
                          Weight: in out Weight_Record);
```

In this case, the expression clearly results in a Weight, an abstract domain already identified. For uniformity, a row record is used for the output, even though the record contains only one component. The type of the component must allow for nulls, that is, must be `Weight_Type` rather than `Weight_Not_Null`, since, if `PNUMBER` is not the number of some part for which some orders are recorded in `SP`, the result of this query is the null value ([2] Section 5.8, general rule 4.c).

4.1.2. A Note on Naming and Packaging

The SAME does not mandate any specific packaging of abstract interface procedures. As mentioned, the rules of SQL require the concrete interface to be a single package. The abstract interface can be partitioned as fits the needs of the application. To prevent unnecessary recompilations, the concrete interface should be imported into the context of the bodies, not the specifications, of the abstract module packages.

In general, the SAME does not specify the names of the procedures at the abstract interface nor the names of their parameters. This naming is the responsibility of the application builder. However, the SAME suggests that the set of procedures associated with a given cursor declaration, the OPEN, FETCH, CLOSE and if needed, positioned UPDATE and DELETE procedures, be placed in a separate package or subpackage of the abstract interface. The name of the package can be the name of the cursor. The open procedure for a given cursor, for example, is then referred to as CURSOR_NAME.OPEN.

4.2. Constructing an Abstract Module

The bodies of the procedures declared in the abstract interface form the abstract module. Each of these procedure bodies has much the same form.

1. The concrete module procedure is called.
2. The status code field (SQLCODE) is processed according to the procedures described in Section 4.3.
3. Type conversions are applied to the parameters at the concrete interface, transforming them to objects of the types at the abstract interface.

For procedures that take input parameters, step 3 occurs first and in the other direction. If a procedure takes no parameters, step 3 does not occur at all. The type conversions of step 3 generally take the form of a test for null, followed by an Assign procedure call.

Example

The body of the procedure Calculate_Weight (of the prior example) is displayed in Figure 4-2, with the package declarations and context clauses omitted for brevity.

The input parameter, PNUMBER, must be converted to a type (Char) defined in SQL_Standard, using an Ada explicit type conversion. Had PNUMBER had type PNO_Type, a call to Without_Null would be necessary and Null_Value_Error might be raised. The concrete module, as given earlier, made no provision for null values in PNUMBER, there being no INDICATOR for it. The raising of an exception here conforms to ANSI specifications *Database Language - SQL*, Sections 8.6 and 8.10, general rule 8) for this situation.

The processing of the output is typical. A negative indicator value indicates a null value. A non-null value must be transformed, using an explicit Ada type conversion, from a type in SQL_Standard (in this case, Int), to the _Not_Null type and then, if necessary, to the output type, via With_Null.

```

procedure Calculate_Weight (PNUMBER : in PNO_Not_Null;
                           Weight : in out Weight_Record) is
Weight_Indic : SQL_Standard.Indicator_Type;
Weight_Buffer : SQL_Standard.Int;
begin
  Concrete_Mod.Calculate_Weight (
    SQL_Standard.Char(PNUMBER),
    Weight_Buffer,
    Weight_Indic,
    SQL_Communications_Pkg.SQLCODE);
  If SQL_Communications_Pkg.SQLCODE /= 0 then
    <see section 4.3>
  end if;
  If Weight_Indic < 0 then
    assign(Weight.Total, Null_SQL_Int);
  else
    assign(Weight.Total,
           With_Null(Weight_Not_Null(Weight_Buffer)));
end Calculate_Weight;

```

Figure 4-2: The Abstract Module Procedure Calculate_Weight

4.3. Database Exceptional Conditions

Every database interaction is capable of failing. Application programmers frequently forget this, and assume that some database interaction will always succeed. Frequently, they assume that a given interaction can fail in one of a small set of predictable ways (e. g., no record found) and forget to check for unpredictable, unrecoverable failures (e. g., disk errors). The net result is that in the presence of failure, the application program behaves in ways that cannot be predicted or analyzed. The SAME provides a robust treatment of database exceptional conditions which allows the average application to assume a failure free database while allowing more sophisticated applications the freedom to do their own error recovery.

ANSI standard SQL systems signal the presence of an exceptional condition through a status parameter called SQLCODE. The values of this parameter are not set by the standard and therefore differ from implementation to implementation. The number of distinct error values is usually in the hundreds. The overwhelming majority of these values signal fatal errors from which the average application will not be able to recover. The SAME is oriented to the needs of such an average application.

The following steps constitute the treatment of database exceptional conditions in the SAME:

1. As each SQL statement is designed and written for the application program, the set of DBMS error conditions which the application will tolerate must be identified. In the most frequently occurring cases, this set will either be empty or will be the singleton "no record found" condition.
2. If the set identified in the prior step is not empty, the abstract interface specification of the procedure that executes that statement will include the optional result parameter. That parameter has an enumeration type, frequently, but not necessarily, BOOLEAN. If the application is sensitive to failure but not to failure mode (or in the case that the set identified above is a singleton), a Boolean is sufficient. The mapping of status code values to enumeration values must be

determined. (For example, a "no record found" condition returned from a DELETE may be considered a successful termination.)

3. The body of this procedure in the abstract module body will then, upon return from the concrete procedure, examine the SQLCODE variable (see Section 4.3.1). The value of the result parameter is set correctly, in the case of success or of a failure mode anticipated in the set described above. In the case of a failure mode outside that set, the procedure *Process_Database_Error* declared in package *SQL_Database_Error_Pkg* is called and the exception *SQL_Database_Error* declared in *SQL_Communications_Pkg* is raised.

This treatment allows the application programmer to ignore exceptional database conditions that are not germane to the application and from which it cannot recover. Raising an exception makes the condition difficult, although not impossible, to ignore. When desired, an error recovery routine can be coded as a handler for the *SQL_Database_Error* exception.

4.3.1. The Packages *SQL_Communications_Pkg* and *SQL_Database_Error_Pkg*

The SAME standard packages *SQL_Communications_Pkg* and *SQL_Database_Error_Pkg* support the authors of abstract modules and of those applications which do more sophisticated error recovery processing. The specification of these packages can be found in Figure 4-3. Both of these packages must be tailored by the user. The specifications in Figure 4-3 are the basic skeletons, which may be modified as needed.

SQL_Communications_Pkg is specific to the platform; it must be tailored to the specific DBMS in use at a site. There need be only one copy of *SQL_Communications_Pkg* at a site. *SQL_Database_Error_Pkg* is specific to the application. There may be more than one copy of this package at a site. In the most likely case, many applications will share a copy of *SQL_Database_Error_Pkg*. The package is best described as being specific to an application class.

Every module language procedure must contain an *<sqlcode parameter>* (*Database Language - SQL*, Section 7.3, syntax rule 6). The call to each concrete module procedure from each abstract module procedure uses the global variable *SQLCODE* declared in the specification of *Sql_Communications_Pkg*.²⁶ Given the importance of the status code, it is best not to duplicate it unnecessarily as that could lead to confusion over which copy is current. (Only the most recent value of the status code is of interest.)

The procedure *Process_Database_Error* should perform whatever processing must be done before the exception is raised and information is lost. This procedure should not attempt error recovery. That should be done by the exception handler. Rather, this procedure gathers whatever information will be needed by the recovery mechanism. It is legitimate, and probably desirable, for *Process_Database_Error* to initiate a transaction rollback. For that to be the case, the procedure must be able to find, (that is know the name of) a sub-program that will cause the SQL ROLLBACK WORK command to be executed.

²⁶Most DBMSs define a communications area which includes a good deal of information beyond *SQLCODE*. The SAME allows for modifications of the specification of *SQL_Communications_Pkg* to include that information. Populating those variables with data is a DBMS-specific task, not covered by the SAME.

SQL_Communications_Pkg

```
with SQL_Char_Pkg, SQL_Standard;
use SQL_Char_Pkg, SQL_Standard;
package SQL_Communications_Pkg is

    SQL_Database_Error : exception;

    SQLCODE : SQLCODE_TYPE; -- Global variable

    -- parameterless function returning an error message of type
    --   SQL_Char_Not_Null
    -- the error message is the descriptive string associated with
    -- the most recent database error

    function SQL_Database_Error_Message return SQL_Char_Not_Null;

end SQL_Communications_Pkg;
```

SQL_Database_Error_Pkg

```
package SQL_Database_Error_Pkg is

    -- The following procedure must be present in every version of
    -- SQL_Database_Error_Pkg. It's purpose is to perform standard
    -- processing of unexpected exceptional conditions. It should not
    -- attempt error recovery.

    procedure Process_Database_Error;

end SQL_Database_Error_Pkg;
```

Figure 4-3: Package Specifications for Sql_Communications_Pkg and SQL_Database_Error_Pkg

In the most frequently occurring case, there will be no handler for the SQL_Database_Error exception. The exception is raised only when an exceptional condition from which the application cannot recover arises. Generally, this indicates either a programming error or a corruption of the database. Manual intervention will usually be required to repair the condition that caused the exception to be raised. The purpose of Process_Database_Error is to display a suitable error message on a suitable device or devices so that the nature of the error will be known. The choice of device may depend upon the class of an application. Batch applications may wish to notify the system operator, record the message in an error log and place a copy into the standard application output file. Online applications may do all of those things and also notify the terminal user.

Most SQL DBMSs provide a routine that converts an SQLCODE value into a meaningful message. The function SQL_Database_Error_Message in SQL_Communications_Pkg is meant to interface to that routine. As the ANSI standard does not include this functionality, the body of this function must be tailored to the DBMS.

4.3.2. Handler for SQL_Database_Error

Applications which must be fault tolerant, and applications written in accordance with local standards prohibiting unhandled exceptions, will provide exception handlers for the SQL_Database_Error exception. These handlers typically appear fairly high in the dynamic call structure of the application, e.g., in a driver procedure, as they are meant to deal with errors that are fairly general in nature. Recall that the exception handler deals only with conditions that the application itself could not process.

If an exception handler is to be used in an application, the Process_Database_Error procedure may need to be specialized to work cooperatively with the handler. For example, if the procedure initiates a rollback operation, the contents of the global variable SQLCODE at the time of failure will be destroyed by the rollback operation. It may be that the handler, not executed until after the termination of Process_Database_Error, will obviate the need for the rollback by repairing the error.²⁷ The handler may need information which has been destroyed by the exception's being raised. Process_Database_Error may save such information for the handler's use. (It will have to do so either in global variables, as its local variables will have been destroyed when the handler is run, or by calling subprograms visible to the exception handler which can accept and store the information.) Specializations such as these may require modifications to the specifications of the packages SQL_Database_Error_Pkg and SQL_Communications_Pkg. This is perfectly acceptable, provided that the global variable SQLCODE and the procedure Process_Database_Error appear as shown in Figure 4-3.

As has been stated, the goals of the SAME treatment of the SQLCODE status parameter are:

1. To free the application programmer from any concern with exceptional conditions not meaningful to the application.
2. To make the occurrence of such exceptional conditions known to the people running the application and difficult for the application to ignore in order to prevent the eventual application failure from being unanalyzable.
3. To allow fault-tolerant programs the ability to recover from system failures.

It is possible for a software development organization to meet these goals through the promulgation of programming standards. The SAME treatment of the SQLCODE parameter ensures that errors are handled in a standard manner specified by the organization, without the need for standards enforcement. This is because the realization of those standards lies not with the application programmers, but rather with the system software designers. Most organizations should find that they need very few distinct copies of the packages involved in this processing, which can be shared by the application programs.

²⁷This seems unlikely. More likely is that an exception handler will trap the exception, to prevent abnormal program termination, and allow the application to restart (rather than recover). Since the underlying problem has not been repaired, it may recur.

4.4. Note on the Overloading of INDICATOR Parameters

The primary purpose of indicator parameters in the SQL module language is the indication of null values. (See *Database Language - SQL*, Section 4.10.2.) However, indicator parameters have a secondary usage, described by general rule 8.a of Sections 8.6 and 8.10 of *Database Language - SQL*:

[Let V be an output parameter and v be the non null value to be assigned to V.] If the data type of V is character string of length L and the length M of v is larger than L, then the indicator is set to M.

In other words, indicators can be used to inform the program that a character string has been truncated. Interestingly, if L in the above is larger than M, padding occurs and the program is not informed.

Since the SAME uses Ada's abstract typing facilities to encapsulate null values, it does not support indicators at the abstract interface. The SAME-DC felt that the use of indicators described in the above quotation would be of use to only a small fraction of all database applications. A means of satisfying those applications without penalizing the majority of applications was developed.

An abstract procedure that corresponds to a concrete FETCH or SELECT statement may declare an additional record parameter. This record will have components all of type SQL_Standard.Indicator_Type (or a type derived from this, if desired). Each component of this indicator record corresponds to a string component of the row record. The name of each component in the indicator record is the name of the component in the row record, and they appear in the same order although some string components may be missing. The body of the abstract procedure copies the indicator values from the concrete indicator parameters to the components of the indicator record for those string components that have indicators.

The SAME-DC felt that this solution was the cleanest available. Altering the row record type definitions to include indicators seemed inappropriate. Altering the abstract types, SQL_Char and SQL_Char_Not_Null, would have penalized all applications to support only a few.

5. Notes on Writing Application Programs Using the SAME Method

This chapter contains hints and suggestions for the designer and programmer using the SAME for an Ada database application.

5.1. Design Rules

The SAME method of constructing database applications divides the problem into two parts: the part to be solved in Ada and the part to be solved in SQL. A rule of thumb to use in determining this division is: If a part of the problem can be solved in either the Ada or the SQL portion of the application, solve it in SQL. The rationale for this rule is that the more the database management system knows, the more it can optimize its behavior. For example, suppose an application is interested in all "red" parts. It is possible to write an SQL statement which returns all parts and an Ada program which finds the red ones. However, it is also possible to write an SQL statement which returns only red parts. In that case, at the very least, there will be fewer calls from the Ada application to the DBMS at runtime. If an index on COLOR exists in the database, the total runtime can be drastically reduced.

5.2. Visibility and the Use of *use*

The application program will need visibility to the domain packages that define the relevant types and to the abstract interface. The domain packages have been designed to be **used**. The domain packages contain instantiated subpackages that are likewise meant to be **used**. This use of **use** allows the operators (comparison and arithmetic) defined in the support package to be used in their normal infix notation. These domain packages typically declare, either by generic instantiation or subprogram derivation, numerous versions of subprograms with the same name. These subprograms can be distinguished by their parameter profiles and often can be distinguished only in that way. Giving their complete names will not uniquely identify them.

There is a situation in which **use** should not be used in the SAME. If two subtypes of a type are declared in a domain package and generic subpackages instantiated for them, the subprograms generated in those subpackages will have the same parameter profiles. If only one of the subtypes is needed in the application, it can be **used** in the normal way. However, if both subpackages are **used**, they will effectively hide each other. In this case, neither subpackage should be **used**; subprograms within them should be referred to as <subpackage name>.<subprogram name>. Be careful to use the correct subpackage with the correct subtype (see Section 3.3).

(The instantiated generic package which forms part of the declaration of an enumeration type abstract domain [see Section 3.6] is also not meant to be **used**. Use of the domain package will bring the derived function names into scope.)

Application programs should not have visibility to any of the SAME standard packages. They should depend only on the domain packages and abstract interface packages which have been developed for them.

5.3. Using Non-ASCII Character Sets

The SAME support for character database columns is designed to allow SAME application programs to be portable across machines with different native character sets. As a by-product, SAME applications can eliminate unnecessary character set conversions.

If the character set native to the machine on which a SAME application is running is not ASCII, then `SQL_Standard.Character_Set` is not set to `Standard.Character` (see Figure 2-2). Rather, `SQL_Standard.Character_Set` is a renaming, that is a subtype, of an enumeration type which defines the native character set. String literals over that character set can be formed in the normal way, provided that the name of the enumeration type specifying the character set is in scope. The context in which the literal appears must be sufficient to determine which character set is to be used, since the predefined package `Standard` cannot be taken out of the scope of any Ada compilation unit.

If, for example, the host character set is supported by a package named `Host_Character_Pkg`, then the application can `use Host_Character_Pkg` if it needs to contain string literals over the host characters. Let `String_Var` and `String_Var_Not_Null` be variables of types derived from `SQL_Char` and `SQL_Char_Not_Null`, respectively. If the name of the DBMS character type is in scope, then both

```
Equals(String_Var, With_Null("A String"))
```

and

```
String_Var_Not_Null = "A String"
```

are syntactically correct and behave as expected.

If the character set native to a machine on which a SAME application is to be run is ASCII, that is, if `SQL_Standard.Character_Set` is `SQL_Standard.Char`, then the predefined Ada type string and the type `SQL_Char_Not_Null` (and types derived from it) are structurally identical (they are both unconstrained one dimensional arrays with the same component type), and are interconvertible using Ada explicit type conversions. If such conversions are used, however, the resulting code is not portable to a machine whose native character set is not ASCII. The functions `To_String` (and `To_Unpadded_String`) and `To_SQL_Char_Not_Null` (and `To_SQL_Char`) are modified at the time of SAME software installation to make them aware of the native character set and to properly perform the type conversion. Use of these functions exclusively for the purpose of such conversions results in an application that is portable across machines with different character sets. However, one further step is needed to complete this portability. If the advice given to `use Host_Character_Pkg` to enable string literal formation is followed, the resulting code will not compile on a machine whose native character set is ASCII and on which, presumably, `Host_Character_Pkg` does not exist. To ensure correct behavior on both ASCII and non-ASCII machines, the program should `use` the package `SQL_Standard.Character_Set`. `SQL_Standard` is not meant to be visible to application programs. The package `SQL_Base_Types_Pkg` described in Section 3.8 contains a renaming declaration of that package. Therefore, a character set independent program should `use SQL_Base_Types_Pkg.Character_Set` to enable formation of literals of types derived from `SQL_Char_Not_Null`.

Although one speaks of a given machine's native character set, it is neither the CPU nor the magnetic storage media that are sensitive to character set encodings. These encodings are

properties of the display devices, printers, and terminals attached to the system. In many DBMS applications, character strings are retrieved from the database and displayed on a display device, often without being examined by the software. It is highly inefficient to convert such data from the native character set to ASCII as the data is read from the database, and then from ASCII to the native character set as the data is displayed on the output device. The conversion is time-consuming and does nothing to forward the application's progress. If all character string variables within an application are of types derived from `SQL_Char_Not_Null` (or `SQL_Char`), those conversions will not occur.²⁸

5.4. Handling the `Null_Value_Error` Exception

The exception `Null_Value_Error` is raised by subprograms of the SAME standard packages when an invalid use of a null value is detected. Typically, this is an attempt to convert the null value to a type which does not support nulls. The exception is defined in the SAME standard package `SQL_Exceptions`. In order to provide a handler for that exception, the package must be brought into scope.

5.5. Simulating Predefined Attributes

The limited private types which the SAME standard packages use to simulate SQL data semantics have operations which allow objects of those types, and the types derived from them that appear in abstract domain declarations in domain packages, to appear very much like visible Ada types. For example, variables of the `SQL_Int` types `Weight_Type`, `Status_Type`, and `Qty_Type` (see Figure 3-7) support arithmetic and comparison operators identical to the Ada integer operators whenever the values of those variables are not null. Since the types are limited private, however, the attributes predefined for integer types are not available. Most of those attributes can be simulated.

Those attributes which are properties of the type, rather than properties of objects of the type or functions defined on objects of the type, can be applied to the `_Not_Null` type. That is, `Weight_Type'First` is not defined but `Weight_Not_Null'First` is defined and is the smallest non-null value that can be stored in a variable of type `Weight_Type`.

Many of those attributes which are properties of objects or functions on objects are duplicated by functions defined on the limited private type. Examples of these are `Succ`, `Pred`, `Image`, and `Value` for enumeration types, and `Image` and `Value` for integer types. The length attribute for strings is simulated by the discriminant, `Length`, of the `SQL_Char` type. Recall that the discriminant of a limited type is visible outside the package defining the type. The attributes `'Range`, `'First`, and `'Last` are not simulated for `SQL_Char`, nor is it possible to access individual characters of a string object of a type derived from `SQL_Char`. Suppose, for example, some processing is to be done if a variable `String_Var`, of a null-bearing type derived from `SQL_Char`, contains the character "X." The following code fragment is correct.

²⁸There are Ada contexts in which the predefined type string is mandatory: the subprograms within the package `TEXT_IO` and the parameters of the `'Image` and `'Value` attribute functions. The latter functions are duplicated by functions defined in the SAME support software. The SAME does not provide a replacement for `TEXT_IO`.

```

for i in 1..String_Var.Length loop
  if Is_True(Equals(substring(String_Var,i,1)),
              With_Null("X")) then
    -- process as needed
    exit;
  end if;
end loop;

```

At the expense of a temporary variable assignment, the above code could be rewritten as:

```

String_Var_Not_Null := Without_Null(String_Var);
for i in String_Var_Not_Null'Range loop
  if String_Var_Not_Null(i) = 'X' then
    -- process as needed
    exit;
  end if;
end loop;

```

but this code is correct only if String_Var is known not to be null. The original code is correct, in the sense that the process is executed only if String_Var contains the character "X", in all cases. The following version is robust and more efficient, particularly when the string of trailing blanks in String_Var is long.

```

if Not_Null(String_Var) then
  String_Var_Not_Null := Without_Null(String_Var);
  for i in 1..Unpadded_Length(String_Var) loop
    -- Since String_Var_Not_Null has the _Not_Null type
    -- of some abstract domain, String_Var_Not_Null'First = 1.
    if String_Var_Not_Null(i) = 'X' then
      -- process as needed
      exit;
    end if;
  end loop;
end if;

```

The extended example of Chapter 8 contains further examples of this kind of processing.

5.6. Doing Type Conversions

It sometimes becomes necessary in Ada programs to convert an object from one type to another. This section contains some details to be kept in mind when type converting database objects.

5.6.1. Ada Explicit Type Conversions

For all domains, except those based on a binary coded decimal (BCD) concrete representation, the non-null bearing `_Not_Null` types are visible Ada types. Therefore, type conversion for objects of these types is done in the ordinary way. The null bearing `_Type` objects are of a limited private type. (This is also true of the `_Not_Null` decimal objects.) Objects of these types are interconvertible with other objects derived from the same base type, directly or indirectly. This is to say, any object the type of which is based on `SQL_Int` can be converted by an Ada explicit type conversion to any other type based on `SQL_Int`. Such an object cannot be converted by such a conversion to an object of a `_Type` derived from `SQL_Smallint`, `SQL_Real`, etc. The following code fragment demonstrates a conversion of an object of a null bearing type derived from `SQL_Int` to an object of a null bearing type derived from `SQL_Real`. (It assumes appropriate visibility.)

```

if Is_Null (Integer_Object) then
    Assign(Real_Object, Null_SQL_Real);
else
    Assign(Real_Object,
        With_Null(Real_Object_Not_Null(Without_Null(Integer_Object))));
end if;

```

(Real_Object is assumed to be of type Real_Object_Type. Real_Object_Not_Null is the corresponding non-null bearing type.)

Special care must be taken when the objects involved are of a character or decimal domain class. These domain class declarations contain subtypes which serve to introduce constraints, string lengths for character domains, and scale for decimal domains. If the subtype names are used as the typemarks for the explicit type conversions, then the domains involved (that is, the source and target domains of the conversion) must specify the same value for these constraints. The procedures for these domain classes allow for inter-type operations. For example, the character Assign will change the length of a string object, padding with blanks or truncating silently; the decimal Assign will change scale, rounding when scale is decreased, providing zeroes when scale is increased. To access this functionality and prevent runtime errors, use the type names of the domain declaration rather than subtype names. (These have the suffix _Base rather than _Type. Note: These rules apply to decimal objects and null bearing character string objects. Non-null bearing character string objects are visible, one dimensional Ada arrays. The standard rules of Ada assignment apply to them.)

5.6.2. Using Conversion Functions

The support for integer and decimal types in the SAME includes functions that convert between objects of those types and objects of unrelated types. (All abstract domains have functions that convert between the null bearing and non-null bearing types within the domain definition.) There is no such support for the floating point types. For the integer types, this support consists of the Image and Value functions. These are semantically equivalent to the 'Image and 'Value predefined attributes for integer types, but their character string operands are over the database character set; that is, they take or return objects of type SQL_Char or SQL_Char_Not_Null defined in SQL_Char_Pkg. Applications do not have visibility to that package and cannot directly declare objects of those types. The package SQL_Base_Types_Pkg, displayed in Figure 3-8, can be used to circumvent this problem.

When taking the Image of a database integer value, the resulting object can be immediately converted to a type visible and meaningful to the application. The following is an example. It is coded within the scope of use clauses for SQL_Base_Types_Pkg, SQL_Base_Types_Pkg.SQL_Char_Ops, Parts_Definition_Pkg, and Parts_Definition_Pkg.Weight_Ops.

```

Integer_As_Character_Object : SQL_Char_Type(SQL_Int_Not_Null'Width);
Weight_Object : Weight_Type;
...
begin
    Assign(Integer_As_Character_Object, SQL_Char_Type(Image(Weight_Object)));
    ...
end;

```

Notice the use of the 'Width attribute of the database integer type to set the length of the output type as large as needed. Since Weight_Object is of the null bearing Weight_Type, the Image function applied to it returns an object of the null bearing type

SQL_Char_Pkg.SQL_Char. This is immediately converted to the visible type SQL_Base_Types_Pkg.SQL_Char_Type. The proper overloading of the Assign procedure, in SQL_Base_Types_Pkg.SQL_Char_Ops, is then found by the compiler. (The base type SQL_Char_Type was used for Integer_As_Character_Object under the assumption that it serves a general role of preparing values for display, rather than a role specific to weights.)

In order to execute the Value function to perform the inverse conversion, the operand must be converted to the appropriate character base type. The subtype names defined in SQL_Base_Types_Pkg can be used as typemarks for this conversion. The inverse of the assignment above is:

```
Assign(Weight_Object, Value(SQL_Char_Subtype(Integer_As_Character_Object)));
```

The decimal support package provides an extensive collection of conversion functions. These convert between the database integer, floating point and character string types, both null and non-null bearing, and the null and non-null bearing decimal types. Use of these conversion functions follows the pattern described for Image and Value. Functions which convert to the other (non-decimal) types are called within the context of a type conversion to a locally visible, appropriate type. Functions which convert from those types to a decimal type take operands which are of the form of a type conversion to the appropriate base type, using the subtypes declared in SQL_Base_Types_Pkg as the typemark. For example, suppose Integer_Object is of a type derived from SQL_Int_Not_Null and its value is to be assigned to Decimal_Object, of a type derived from SQL_Decimal. The following Assign procedure call accomplishes this:

```
Assign(Decimal_Object,  
       To_SQL_Decimal(SQL_Int_Not_Null_Subtype(Integer_Object)));
```

5.7. Using Three-Valued Logic

The SAME's treatment of null values (see Section 3.1) replicates the SQL semantics. Database objects which might be null can be operated on with arithmetic and comparison operations in place. They do not have to be converted to visible Ada types. To do this successfully, however, the programmer must understand SQL semantics for the null value.

Briefly, any operator that is not a conversion function, other than comparisons, returns the null value when at least one of its inputs is the null value. The comparison operators return the truth value UNKNOWN if one of the comparands is the null value.

The SQL null value represents missing or unknown information. The expressions "2 + null" means "add two to an unknown number." The answer is an unknown number, that is, the null value. Similarly, the comparison "2 > null" means "is two greater than an unknown number." The answer is the new truth value, UNKNOWN.

When using SQL arithmetic, the programmer or analyst must decide whether the null answer is acceptable. The null answer indicates that some of the input was missing and that an accurate calculation is impossible. If the null answer is not acceptable, then a strategy for dealing with null values in the input must be chosen. SQL will filter out null values, but this may not be acceptable within the context of the application, because it may cause other information to be lost. Null values can be detected with the Is_Null and Not_Null Boolean-valued functions that every SAME standard package exports. The application must decide what to do with those values.

SQL arithmetic and three-valued logic are most useful in short calculations leading to tests. For example, suppose a process is to be applied in case a Status variable (of type Status_Type, which may be null) has a value in excess of one hundred. This can be written as:

```
if Status > With_Null(100) then
    <perform process>
end if;
```

The operator ">" is resolved to the Boolean-valued operator taking objects of type Status_Type which operator is created as part of the derivation of Status_Type from SQL_Int_Pkg.SQL_Int. This operator returns "false" if either operand is null. Were the process to be applied in case Status *might* be in excess of one hundred, it would be written as:

```
if not (Status <= With_Null(100)) then
    <perform process>
end if;
```

or as:

```
if not Is_False(Status > With_Null(100)) then
    <perform process>
end if;
```

In either case, the process is performed for a Status value of null, as well as known values over one hundred.

Three-valued logic can be most helpful in evaluating compound predicates. One can think of the versions of **or** and **and** exported by SQL_Boolean_Pkg as being symmetric versions of Ada's **or else** and **and then**. Thus the process in this statement

```
if Is_True(Status > With_Null(100) or
           Equals(City, With_Null("Pittsburgh"))) then
    <perform process>
end if;
```

will be performed if at least one of the two conditions is known to be true. Unlike Ada's **or else**, the first condition may be non-computable, that is, UNKNOWN, and the second True. The example can also be written as:

```
if Status > With_Null(100) or else
   City = With_Null("Pittsburgh") then
    <perform process>
end if;
```

in which case, the second comparison will not be made if the first comparison returns "true."

The package SQL_Boolean_Pkg defines the type Boolean_With_Unknown and the functions which operate on it. The application program must have visibility to that package to use those functions. As discussed above, the package is meant to be **used**.

5.8. Commenting Procedure Calls

To improve the readability of SAME applications, it is good practice to annotate the calls to abstract interface procedures with an English description of the call's effect. This annotation should also appear on the declaration of the procedure in the abstract interface. It is bad practice to use the SQL statement as the annotation. An advantage of the SAME is that the SQL statements in the concrete module can be modified without modification, indeed, without recompilation, of the application. Further, proper understanding of the SQL statement requires an understanding of the database structure and semantics. If the comment is in

English and not in SQL, it may be understood by readers who are ignorant of the database structure.

The SQL statement as comment may be very uninformative. The SQL `FETCH` statement says very little about what is being fetched. In so far as that is present in the concrete module, it is the associated `DECLARE CURSOR` statement. It is better to use an English description such as "retrieves the next pair of part numbers and cities meeting the run time restriction on supplier status" (see the example in the introduction) rather than "fetch x into Part_Number, City INDICATOR City_Indic."

It is likewise good practice to comment the definition of a row record type with an explanation as to the meaning of objects of the type. This practice is illustrated in the examples of Chapter 8.

6. The SAME Method Summarized

The SAME is a modular approach to Ada SQL interfacing that builds on the capabilities of the ANSI standard module language. The value added by the SAME beyond the module language itself includes:

- a safe treatment of null values
- a robust treatment of exceptional conditions
- full Ada typing
- decimal arithmetic in Ada
- SQL string operations in Ada
- extensibility to data types not in the SQL standard (such as Ada enumeration types)

There exist standard SAME packages which implement these features. They appear in Appendix C of this report. This support includes an implementation of three-valued logic which conforms to SQL definitions.

The SAME is used in the following way:

- During the database design process the abstract domains occupying the database columns must be identified and described as Ada types. These type definitions are stored as domain packages.
- During the design of an application, the services needed from the database are identified and coded as SQL statements. They are collected into a module. This is called a *concrete module*.
- For each data item at the abstract interface, the type within the abstract domain definition for that item must be determined. If the data item is logically capable of taking on the null value, an Ada type capable of taking on a null value, e. g., the *_Type* rather than the *_Not_Null* type, must be used.
- An abstract interface is created. This is a set of package specifications declaring whatever record type definitions are needed to describe row records and whatever procedure declarations are needed to access the relevant concrete module procedures.
- The abstract module, the bodies of the procedures declared in the abstract interface, is created. The procedures in the abstract module have the following structure:
 1. The corresponding concrete procedure is called; the global parameter *SQLCODE* in the package *SQL_Communications_Pkg* is used as the *<sqlcode parameter>*.
 2. The *SQLCODE* value is processed as appropriate. When unanticipated errors occur, a standard routine, *Process_Database_Error* in the package *SQL_Database_Error_Pkg*, is called. This routine is specialized to a class of applications, e.g., batch, online, etc. Upon return from that routine, the exception *SQL_Database_Error* is raised.

3. Assuming the exception is not raised, data values are examined for null (indicator values) and assigned to output parameters for type conversion and range checking. (If data is flowing from the application to the database, as for UPDATE and INSERT commands, this step occurs first. If data is flowing in neither direction, as for e.g., *close*, this step is omitted.)

- The application program can be written while the abstract module is being written. It will need access to the relevant domain packages and to the abstract interface. It can treat incomplete information (null values) in either a "test and convert" fashion or with the full three-valued logic and arithmetic of SQL. It can ignore all database errors from which it cannot recover.

Figure 6-1 diagrams the package structure of a complete SAME application. Although only one domain package and abstract interface module are shown, these may be divided into multiple packages at the designer's discretion. The shaded areas indicate those parts of an application which are unique to it. The arrows represent visibility (**with**) relationships, not call structure. The dashed arrows indicate optional visibility. An application needs visibility to SQL_Boolean_Pkg and SQL_Exceptions only if it executes three-valued Boolean operations or provides an exception handler for the Null_Value_Error exception, respectively.

The packages within the support layer are in the SAME standard packages and are never modified. The package SQL_Database_Error_Pkg may be specialized for classes of applications. The packages SQL_System, SQL_Standard, and SQL_Communications_Pkg are specialized for the DBMS being used.

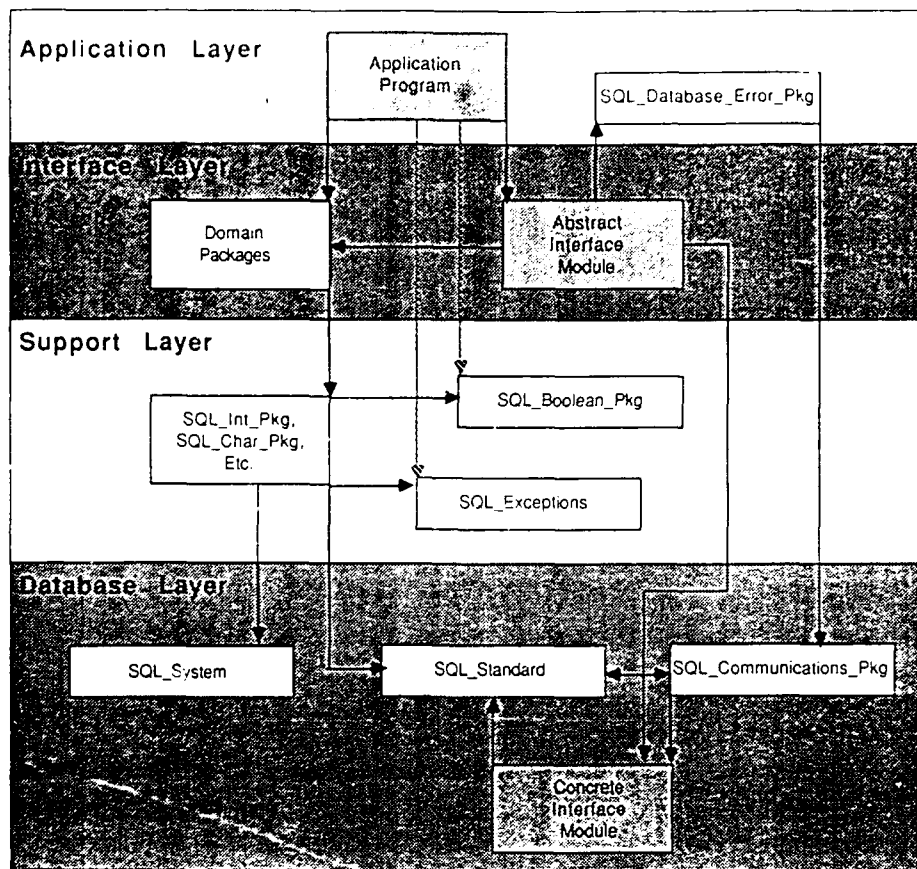


Figure 6-1: SAME Application Package Structure



7. Building a SAME Application Without a Module Compiler

The presentation of the SAME in these guidelines has assumed the existence of a compiler for the module language. The SAME can be used in environments for which no such compiler exists. All that is needed is DBMS support for some programming language. With such support, the module language compiler can be simulated.

The simulation of the module language compiler need not be exact. If the DBMS vendor supplies an SQL preprocessor for Ada, it is reasonable to use it and put SQL statements in place of the calls to the concrete procedures in the bodies of the procedures in the abstract module. The division into abstract and concrete modules is not an essential part of the SAME. It is used primarily for purposes of exposition. It is the interface to the application, the abstract interface, which is the hallmark of the SAME.

If the DBMS vendor supplies no support for Ada, but supplies support for other programming languages, those foreign language processors can be used in place of the module language compiler. This is easiest if the DBMS vendor allows database access from a language to which the Ada compiler interfaces.

The details of foreign language calls are compiler dependent. In general terms, a procedure declaration is followed by a **pragma INTERFACE** statement indicating that the procedure is coded in a foreign language. This pragma may appear in the body of abstract module procedures. When using a foreign language, it is not essential that the concrete module appear as an object.

Example

The example `Concrete_Mod` displayed earlier is repeated here coded in C. It is shown in Figure 7-1 with its Ada call coded for an Alsys Ada compiler (Release 3.0, running on a Sun) [1]. In Figure 7-2 it is shown for a Verdix compiler (Release 5.41, running on a VAXStation) [17]. Both examples are written for Ingres Release 5.0.²⁹

²⁹Ingres 5.0 does not support null values. Therefore, the indicator parameters are missing from the SQL statements.

Concrete_Mod in 'C' for Alsys

```
exec sql include sqlca;

ingcalc (pnumber, totalw, sqlcode)
exec sql begin declare section;
    long pnumber;
    long *totalw;
exec sql end declare section;
    long *sqlcode;
{
exec sql select sum (qty*weight)
into :*totalw
from p, sp
where p.pno = sp.pno
and p.pno = :pnumber;
*sqlcode = sqlca.sqlcode
}
```

The Alsys Ada declaration

```
procedure Calculate_Weight (PNUMBER : SQL_Standard.Int;
    Total_Weight : out SQL_Standard.Int;
    SQLCODE : out SQL_Standard.SQLCODE_Type);
pragma INTERFACE (c, Calculate_Weight);
pragma Interface_Name (Calculate_Weight, "ingcalc");
```

Figure 7-1: Concrete_Mod for Alsys

Concrete_Mod in 'C' for Verdix

```
exec sql include sqlca;

ingcalc (pnumber, totalw, sqlcode)
exec sql begin declare section;
    long *pnumber;
    long *totalw;
exec sql end declare section;
    long *sqlcode;
{
exec sql select sum (qty*weight)
into :*totalw
from p, sp
where p.pno = sp.pno
and p.pno = :*pnumber;
*sqlcode = sqlca.sqlcode
}
```

The Verdix Ada Declaration

```
procedure Calculate_Weight (PNUMBER : System.Address;
    Total_Weight : System.Address;
    SQLCODE : System.Address);
pragma INTERFACE (c, Calculate_Weight, "_ingcalc");
```

Figure 7-2: Concrete_Mod for Verdix

Notice that use of a foreign language makes the abstract module compiler dependent; if the application is moved to a different compiler, the abstract module must be recoded. The abstract interface is not affected; therefore, neither is the application program.

As illustrated in Figures 7-1 and 7-2, the foreign language routines should do only the minimum required. They should contain almost nothing but SQL statements and data declarations. In particular, any differences between the Ada data representation and the foreign language representation should be resolved in the Ada code. For example, C character strings are terminated with the ASCII null. Ada strings are not. The removal and addition of the ASCII null can be done in the Ada abstract module.

One must be careful in using foreign language routines in an Ada program. There is no type checking across the boundary between Ada and the foreign language. Be sure to verify the types by hand. Be sure to leave enough room in character strings to accommodate the ASCII null at the end of C strings, for example.

If the set of languages which the compiler recognizes is disjoint from the set of languages which the DBMS supports, it will be necessary to write an extra interface procedure. This has not been attempted as of this writing; thus, little guidance can be offered.

8. Some Detailed Examples

This section presents an example of the use of the SAME, illustrating features of a SAME application and a SAME abstract module. Details of the application which are irrelevant to the database interaction are not shown; in particular, the details of user interaction are suppressed. Only those fragments of the application which acquire and manipulate database data will be presented.

The design decisions in the examples are contrived to illustrate the coding aspects of abstract modules and application programs. The example should not be taken as an example of good program design.

The example accesses the Parts-Supplier database described in Figure 1-6. The abstract domains describing that database are to be found in Figures 3-6 and 3-7. The overall structure of the application is shown in Figure 8-1. The DRIVER block is responsible for user communication. Based on user input, the DRIVER block determines which application service has been requested and calls the appropriate subprogram, the blocks labeled EXAMPLE_A through EXAMPLE_C in Figure 8-1. The DRIVER program will not be shown. Each of the example blocks has an associated DISPLAY facility which is responsible for displaying the module's results on the user terminals. These display facilities will also not be shown. The complete text of the example subprograms and of the abstract modules will be presented. (This architecture was chosen so that complete subprograms could be shown and irrelevant details could be suppressed.)

Notice that there is only one concrete module in Figure 8-1, labeled EXAMPLE_CONCRETE_MODULE. There are three abstract modules, one for each of the distinct parts of the application. They contain just those database procedures and definitions which are relevant to the application services they support. The bodies of the abstract modules depend on (with) the concrete module. Modifications to and recompilation of the concrete module will, in general, require recompilation of the bodies of the abstract modules, but not their specifications and, therefore, not those parts of the application which are unaffected by the changes to the concrete module.

Example_A

In Example_A, the user enters the number of a part and requests the number of outstanding orders for that part and the total weight of those shipments. The SQL module procedure which retrieves this information is given in Figure 8-2.³⁰ The corresponding abstract module specification is given in Figure 8-3.

The single procedure PartWeight in the Ada abstract module Example_A_Module takes a part number as its single input parameter and returns a record containing the part number, the requested weight and count, and a Boolean result parameter. (The part number is added to the output row record type so that objects of that type have a well defined meaning. The comments on the row record definition in Figure 8-3 give that meaning. It is good practice to comment row record type definitions in this way.) The Boolean takes the value **false** when the requested part number does not have any shipments in the database, in which case the

³⁰Figures containing SQL or Ada code appear at the end of each example.

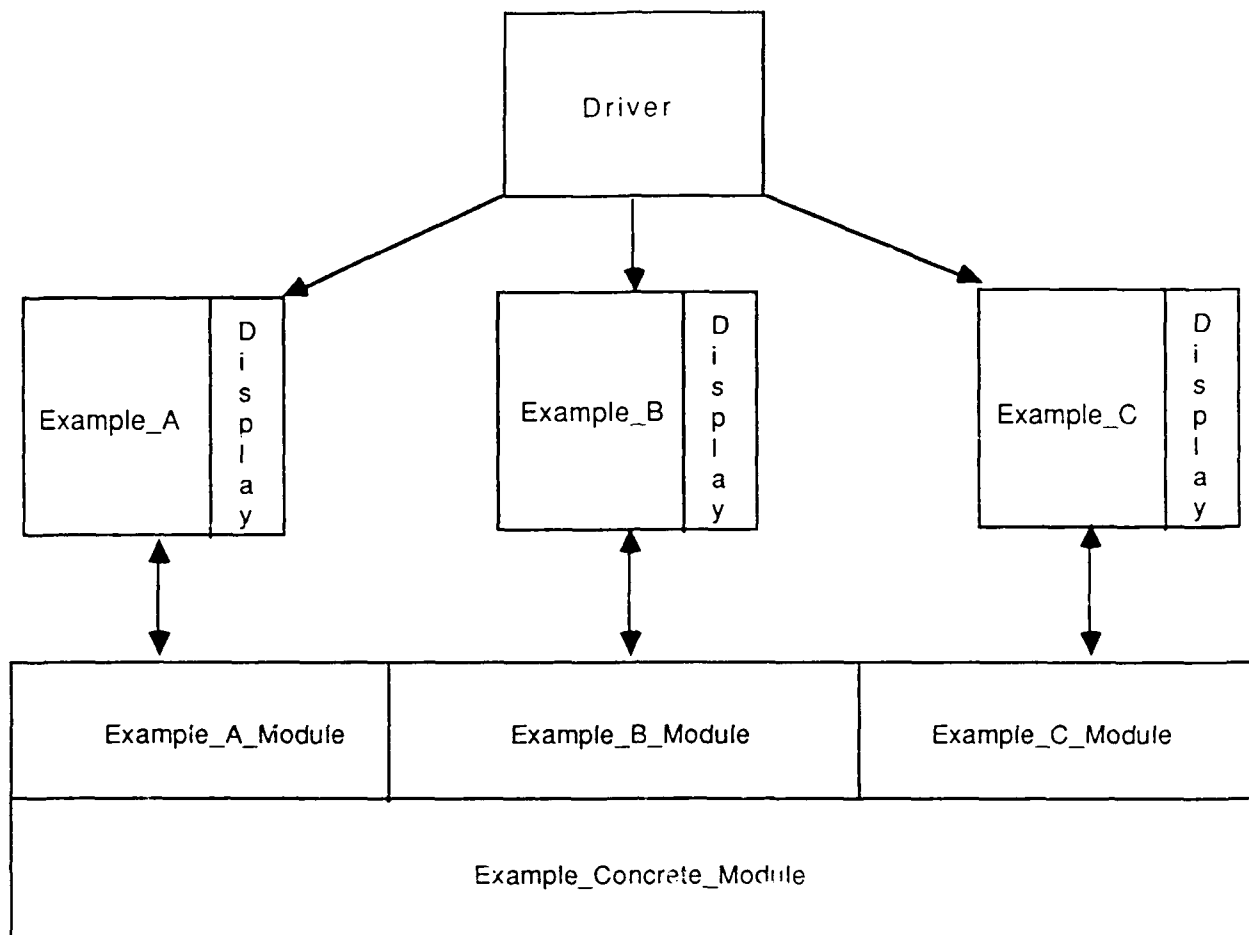


Figure 8-1: A Block Diagram of the Example

value of the record object is unreliable. Although the SQL statement references the quantity (Qty) column of the SP table, the abstract module does not need visibility to the QTY domain defined in QTY_Definition_Pkg (see Figure 3-6) since no values of the QTY domain are passed across the abstract interface.

The Weight component of the result record takes a null bearing type, Weight_Type, as the value returned from the SQL statement may be null. (It will be null when the Weight column of the P table entry for the given Pno is null.) Notice that the SQL statement has an indicator variable attached to the output target specification for Weight_Out, signaling that a null result is possible. The Count component of the result record takes a non-null bearing type as the corresponding value of the SQL statement cannot be null and therefore does not have an attached indicator variable.

The type of the Count component, SQL_Int_Not_Null, is one of the "base domains" defined in the package SQL_Base_Types_Pkg. The package is described in Chapter 3.8.

The bulk of Example_A reformats the database input for the purpose of display. The details of the communication with the display device, including screen formats, are hidden in the

separately compiled subprogram `Display_The_Line_A`. Among other things, `Example_A` must convert integers into character strings. It uses the SAME function `Image`, not the Ada predefined attribute function `'Image`, as the former returns its value in the underlying machine character set whereas the latter returns its value in ASCII. In this and the other examples, each item to be displayed has an associated length field. (The component `Pno` of the `Display_Line` type does not have a length field, as this component is fixed length. The other fields have an associated length, as the length of an integer's image depends on its value.) Because the `Count` component within the `Weight_Count_Record` has a `_Not_Null` type, the `Image` function applied to it returns a character string of the unconstrained array type `SQL_Char_Pkg.SQL_Char_Not_Null`. The length of that string is returned by the `'Length` predefined Ada attribute. The `Weight` component has a null bearing `_Type`, so the `Image` function applied to it returns an object of the limited, discriminated type `SQL_Char_Pkg.SQL_Char`. The length of that object is the value of the discriminant, `Length`. The character strings themselves must be converted to the type `SQL_Base_Types_Pkg.SQL_Char_Not_Null`. These conversions should consume no runtime resources. (This usage of the `SQL_Char` domain in `SQL_Base_Types_Pkg` is illustrative of interfaces to low-level services. Section 3.8 discusses these services and various strategies for using them.)

The body of `Example_A`'s abstract module is presented in Figure 8-6. Its structure is typical of abstract procedures whose SQL statement is a `<select statement>` (`SELECT ... INTO`). Since concrete procedures use the types in `SQL_Standard` as parameter types, the input and output part numbers must be converted, using an Ada-explicit type conversion, to `SQL_Standard.Char`. (Notice that the output part number is deposited directly into the application's buffer from the concrete module's output. Every component of a row record object must be set from a parameter of the concrete module, even in a case like this one, in which an output value is by definition identical to an input value.) This conversion consumes no runtime resources. After the concrete procedure is called, the `SQLCODE` value is analyzed according to the needs of the application. Condition codes other than `Not_Found` or successful completion (zero) invoke standard error processing.

If the input part number exists in the database, the data returned must be converted to the abstract application types. Since the `Count` component of the output is a `_Not_Null` type, that is, a visible Ada integer type, the value returned from the concrete module can be deposited directly in the output component. Thus, with respect to the `Count` component, the abstract module introduces no runtime overhead.

Since the `Weight` component may be null, the abstract module must examine the indicator variable for weight to determine if the actual value is null. The package `Conversions` was written to facilitate this. Its specification and body are presented in Figure 8-7. The use of the `Convert` functions declared in package `Conversions` simplifies the writing of abstract module bodies. Those functions return objects of the base null bearing types, `SQL_Int`, `SQL_Char`, etc. Abstract modules do not have visibility to the packages in which those types are declared, for reasons discussed in Section 3.8. Thus the values returned by these functions must be immediately converted to the output abstract type, as shown. (Notice the use of `pragma Inline` in package `Conversions` to eliminate the expense of a procedure call.)

```

PROCEDURE PartWeight
  Pno_In Char (5)
  Pno_Out Char (5)
  Weight_Out Int Weight_Indic Smallint
  Count_Out Int
  SQLCODE;

  SELECT DISTINCT P.Pno, Weight * Sum(Qty), Count(SP.SNO)
  INTO Pno_Out
      Weight_Out INDICATOR Weight_Indic,
      Count_Out
  FROM P, SP
  WHERE P.Pno = SP.Pno and P.Pno = Pno_In;

```

Figure 8-2: The SQL Procedure for Example_A

```

with SQL_Base_Types_Pkg, Parts_Definition_Pkg;
use SQL_Base_Types_Pkg, Parts_Definition_Pkg;
package Example_A_Module is

  type Weight_Count_Record_Type is record
    Pno : Pno_Not_Null;      -- all the shipments for this part
    Weight : Weight_Type;    -- have this combined weight.
    Count : SQL_Int_Not_Null; -- there are these many
  end record;

  procedure PartWeight (Pno : in Pno_Not_Null;
    Weight_Count : in Out Weight_Count_Record_Type;
    Exists : out boolean);

    -- the result weight is the combined gross weight
    -- of all shipments of the input Weight
    -- Exists is False when Pno not in database

end Example_A_Module;

```

Figure 8-3: The Abstract Module for Example_A

```

with Parts_Definition_Pkg, SQL_Base_Types_Pkg, Example_A_Module;
use Parts_Definition_Pkg, SQL_Base_Types_Pkg, Example_A_Module;
separate (Driver)
procedure Example_A (Pno : Pno_Not_Null) is
use SQL_Char_Ops, SQL_Int_Ops; -- Base type subpackages
use Character_Set;           -- For literal formation
-- literals for display
  No_Data : constant SQL_Char_Not_Null :=
    "Part Number Not in Database";
  Null_Weight : constant SQL_Char_Not_Null :=
    "Null Weight";
-- types used for display
  type Message_Type is (Error_Msg, Data_Msg);
  type Display_Line (Message : Message_Type) is record
    Pno : SQL_Char_Not_Null(Pno_Not_Null'Range);
    case Message is
      when Data_Msg =>
        Weight_Length, Count_Length : Integer;
        -- these are lengths of the data in the
        -- next two fields, which are declared to be
        -- of a maximum length, which in most cases is
        -- much too large
        Weight : SQL_Char_Not_Null(1 ..
          Weight_Not_Null'Width);
        Count : SQL_Char_Not_Null(1 ..
          A_Database_Integer_Not_Null'Width);
      when Error_Msg =>
        -- when the part number doesn't exist, this
        -- variant is used
        Msg : SQL_Char_Not_Null(No_Data'Range) := No_Data;
    end case;
  end record;
-- objects used for display
  Data_Line : Display_Line(Message => Data_Msg);
  Error_Line : Display_Line(Message => Error_Msg);
-- objects used for communication with Abstract Module
  Tuple : Weight_Count_Record_Type; -- holds the output
  Is_Found : Boolean;
-- the display procedure, which will not be shown
  procedure Display_The_Line_A (Line_To_Display : Display_Line)
    is separate;

```

Figure 8-4: Example_A (Part I)

```

begin
  PartWeight (Pno, Tuple, Is_Found);    -- The Abstract procedure
  if Is_Found then                    -- Part Nbr good; prepare output
    Data_Line.Pno := SQL_Char_Not_Null(Tuple.Pno);
    Data_Line.Count_Length := Image(Tuple.Count)'Length;
    Data_Line.Count(Data_Line.Count'First ..
      Data_line.Count'First + Data_Line.Count_Length - 1)
    :=
      SQL_Char_Not_Null(Image(Tuple.Count));
    if Not_Null(Tuple.Weight) then -- for non null weights
      -- prepare outputs
      Data_Line.Weight_Length := Image(Tuple.Weight).Length;
      Data_Line.Weight(Data_Line.Weight'First ..
        Data_Line.Weight'First + Data_Line.Weight_Length - 1)
      :=
        Without_Null(SQL_Char_Type(Image(Tuple.Weight)));
    else
      -- for null weights, prepare a message
      Data_Line.Weight_Length := Null_Weight'Length;
      Data_Line.Weight := Null_Weight;
    end if;
    Display_The_Line_A(Data_Line); -- put out a line of data
  else
    -- the Part Nbr not in DB
    Error_Line.Pno := SQL_Char_Not_Null(Pno);
    Display_The_Line_A (Error_Line); -- a message about missing Part
  end if;
end Example_A;

```

Figure 8-5: Example_A (Part II)


```

with Conversions, SQL_Standard, SQL_Communications_Pkg,
    SQL_Database_Error_Pkg;
use Conversions, SQL_Standard, SQL_Communications_Pkg,
    SQL_Database_Error_Pkg;
with Example_Concrete_Module;
package body Example_A_Module is

    package Conc renames Example_Concrete_Module;

    use Weight_Ops;

    procedure PartWeight (Pno : In Pno_Not_Null;
        Weight_Count : In Out Weight_Count_Record_Type;
        Exists : Out boolean) is

        Weight_Temp : Int;
        Weight_Indic : Indicator_Type;

    begin

        Conc.PartWeight (Char (Pno),
            Char (Weight_Count.Pno),
            Weight_Temp, Weight_Indic,
            Int (Weight_Count.Count),
            SQLCODE);

        If SQLCODE in Not_Found then -- no such part no
            Exists := false;
        elsif SQLCODE /= 0 then -- unrecoverable error
            Process_Database_Error;
            raise SQL_Database_Error;
        else
            Exists := true; -- record retrieved as expected
            Assign (Weight_Count.weight,
                Weight_Type (Convert (Weight_Temp, Weight_Indic)));
        end If;
    end PartWeight;
end Example_A_Module;

```

Figure 8-6: The Abstract Module Body for Example_A

```

with SQL_Standard, SQL_Int_Pkg, SQL_Smallint_Pkg,
     SQL_Char_Pkg, SQL_Real_Pkg,
     SQL_Double_Precision_Pkg;
use SQL_Standard, SQL_Int_Pkg, SQL_Smallint_Pkg,
     SQL_Char_Pkg, SQL_Real_Pkg,
     SQL_Double_Precision_Pkg;

package Conversions is
  function Convert (Input : Int; Indicator : Indicator_Type)
    return SQL_Int;
  function Convert (Input : Smallint; Indicator : Indicator_Type)
    return SQL_Smallint;
  function Convert (Input : Char; Indicator : Indicator_Type)
    return SQL_Char;
  function Convert (Input : Real; Indicator : Indicator_Type)
    return SQL_Real;
  function Convert (Input : Double_Precision; Indicator : Indicator_Type)
    return SQL_Double_Precision;
  pragma inline(Convert);
end Conversions;

package body Conversions is

  subtype Null_Indication is Indicator_Type -- Negative value
    signals Null
    range Indicator_Type'First .. -1;

  function Convert (Input : Int; Indicator : Indicator_Type)
    return SQL_Int is
  begin
    If Indicator in Null_Indication then
      return Null_SQL_Int;
    else
      return With_Null_Base(SQL_Int_Not_Null(Input));
    end If;
  end Convert;

  function Convert (Input : Smallint; Indicator : Indicator_Type)
    return SQL_Smallint is
  begin
    If Indicator in Null_Indication then
      return Null_SQL_Smallint;
    else
      return With_Null_Base(SQL_Smallint_Not_Null(Input));
    end If;
  end Convert;

  function Convert (Input : Real; Indicator : Indicator_Type)
    return SQL_Real is
  begin
    If Indicator in Null_Indication then
      return Null_SQL_Real;
    else
      return With_Null_Base(SQL_Real_Not_Null(Input));
    end If;
  end Convert;

  function Convert (Input : Double_Precision; Indicator : Indicator_Type)
    return SQL_Double_Precision is
  begin
    If Indicator in Null_Indication then
      return Null_SQL_Double_Precision;
    else

```

```

        return With_Null_Base(SQL_Double_Precision_Not_Null(Input));
    end if;
end Convert;
function Convert (Input : Char; Indicator : Indicator_Type)
    return SQL_Char is
begin
    If Indicator in Null_Indication then
        return Null_SQL_Char;
    else
        return With_Null_Base(SQL_Char_Not_Null(Input));
    end if;
end Convert;
end Conversions;

```

Figure 8-7: The Conversions Package

Example_B

Example_B accepts a part number from the user and returns information about each shipment of the part: the part number, the name of the supplier, and the total weight of the shipment. As there are, in general, multiple shipments for a part, a cursor-oriented retrieval is needed. The SQL text of the cursor declaration and its associated procedures is given in Figure 8-8 and the abstract module specification in Figure 8-9. In the abstract module, the cursor procedures appear in a subpackage whose name is the cursor name, Detail. This usage is inessential, in this case, as the abstract module contains only these procedures. For applications which manipulate multiple cursors, the use of abstract module subpackages in this way will improve the readability of the code and prevent name conflicts.

Example_B, which is displayed in Figure 8-10, declares a display-oriented record type containing a variant for part numbers which have no shipments. The body of Example_B opens the cursor, passing the part number into the open procedure, and then retrieves each row of the result, formatting and displaying each of them. Notice that the initial fetch is done outside of the loop, as an end of file condition, for this fetch means the part was not found. Therefore, the loop body first displays the current tuple and then fetches the next tuple. This is a typical paradigm for cursor-oriented database retrieval.

The body of the **while** loop illustrates two new features. The SNAME character string value has its trailing blanks removed by the `Without_Null_Unpadded` function generated by the instantiation of the `SNAME_Ops` subpackage. (Hence, the **use** for that subpackage.) The length of that function result is returned by the `Unpadded_Length` function.

The loop body also contains an example of mixed mode arithmetic. Recall that Example_B returns to the user the total weight of each shipment, the product of the weight of a part, and the quantity of items shipped. This value could have been produced by the SQL statement, which would in reality have been preferable. It was not done in order to illustrate mixed mode arithmetic operations in the SAME.

The quantity value is converted to the weight type, as the target value has weight type. Because the null bearing `_Type(s)` are in use, this Ada explicit type conversion will not produce any runtime exceptions. If the `_Not_Null` types were in use and were range constrained, care would be needed to ensure that a runtime `constraint_error` is not raised.

The body of `Example_B_Module`, the abstract module for Example_B, appears in Figure 8-11. Neither the `Open` nor the `Close` procedures will accept any `SQLCODE` values other than success, e.g., the value zero. These procedures take no result parameter, therefore. The `fetch` procedure signals end of file by returning the **false** Boolean value in its result parameter.

When a tuple is returned, its values must be converted to the application's abstract types. Again the `Pno` value, which cannot be null, is deposited directly into the application's buffer. The values of those items which may be null are read into intermediate variables in the abstract module's data space. They are tested for null and converted to the application's types using the `assign` and `convert` functions shown in Example_A's abstract module. Notice the **use** statement for the generic subpackage instantiations of the integer domains, `Weight`, and `QTY`. This **use** statement makes the `assign` procedures for these domains visible.

Again, the values returned by the Convert functions have the SAME base types (SQL_Int, SQL_Char, etc.) and must therefore immediately be converted to the application's types. This is done with an Ada-explicit type conversion. For the character string based SNAME domain, the target of the type conversion is SNAME_Base and not the SNAME_Type subtype. Recall that the definition of a character string domain consists of two type declarations, two subtype declarations, and a package instantiation. The type declarations declare unconstrained types; the subtypes specify the constraint, i. e., the string length. Now if a given value is null, the Convert function will return Null_SQL_Char, an object of type SQL_Char. This object must, of course, have a discriminant constraint (a Length). Since Convert works only with base types, it cannot know how "long" to make this null value. Thus the length of Null_SQL_Char is one. If this object were converted to the subtype SNAME_Type, a constraint_error (discriminant_error) would occur. Since the type SNAME_Base is unconstrained, the type conversion to it avoids the runtime exception.

```

DECLARE Detail CURSOR FOR
  SELECT P.Pno, S.Sname, SP.Qty, P.Weight
  FROM S, P, SP
  WHERE S.Sno=SP.Sno AND P.Pno = SP.Pno and
         P.Pno = Pno_In;

PROCEDURE DetailOpen
  Pno_In Int
  SQLCODE;

  OPEN Detail;

PROCEDURE FetchDetail
  Pno Char (5)
  Sname Char (20) Sname_Indic Smallint
  Qty Int Qty_Indic Smallint
  Weight Int Weight_Indic Smallint
  SQLCODE;

  FETCH Detail
  INTO Pno,
       Sname INDICATOR Sname_Indic,
       Qty INDICATOR Qty_Indic,
       Weight INDICATOR Weight_Indic;

PROCEDURE CloseDetail
  SQLCODE;

  CLOSE Detail;

```

Figure 8-8: The Cursor Declaration and SQL Procedures for Example_B

```

with QTY_Definition_Pkg, Suppliers_Definition_Pkg, Parts_Definition_Pkg;
use QTY_Definition_Pkg, Suppliers_Definition_Pkg, Parts_Definition_Pkg;
package Example_B_Module is

type Detail_Record_Type is record
  Pno : Pno_Not_Null;           -- this part shipped by
  SName : SNAME_Type;          -- this supplier
  Qty : QTY_Type;              -- in this quantity
  Weight : Weight_Type;        -- each part weighs this much
end record;
package Detail is
  procedure Open (pno : in Pno_Not_Null);

  -- Creates a file of Detail_Records for the part
  -- whose number is given

  procedure Fetch (Tuple : in out Detail_Record_Type;
                  Found : out Boolean);

  -- returns the records created by the open
  -- found becomes false at eof

  procedure Close;
end Detail;
end Example_B_Module;

```

Figure 8-9: The Abstract Module for Example_B

```

with Example_B_Module, Parts_Definition_Pkg, Suppliers_Definition_Pkg,
     QTY_Definition_Pkg, SQL_Base_Types_Pkg;
use Example_B_Module, Parts_Definition_Pkg, Suppliers_Definition_Pkg,
     QTY_Definition_Pkg, SQL_Base_Types_Pkg;
separate (Driver)
procedure Example_B (Pno : Pno_Not_Null) is

    use Character_Set, SNAME_Ops, Weight_Ops, SQL_Char_Ops;

    -- literal for error message display
    No_Data : constant SQL_Char_Not_Null := "Part Number " &
        SQL_Char_Not_Null(Pno) & " has no shipments";
    -- Strings For Printing Null values
    Null_Sname : constant SQL_Char_Not_Null := "No Supplier Name";
    Null_Weight : constant SQL_Char_Not_Null := "No Weight";
    -- types for display
    type Line_Type is (Error_Line, Data_Line);
    type Display_Line (Kind : Line_Type) is record
        case Kind is
            when Error_Line =>
                -- this is used when the part has no shipments
                Msgg : SQL_Char_Not_Null(No_Data'Range) := No_Data;
            when Data_Line =>
                -- this is used when the part can be found
                -- each field (except Pno)
                -- has a length field. The field is big enough
                -- for the largest possible value. The length field
                -- contains the size of the actual value.
                Pno : SQL_Char_Not_Null(Pno_Not_Null'Range);
                Sname_Length : integer;
                Sname : SQL_Char_Not_Null(Sname_Not_Null'Range);
                Total_Weight_Length : integer;
                Total_Weight : SQL_Char_Not_Null(1 ..
                    Weight_Not_Null'Width);
        end case;
    end record;

    -- Put the display line out (not shown)
    procedure Display_The_Line_B (A_Line : in Display_Line)
        is separate;

-- body of Example_B
begin
    declare
        Tuple : Detail_Record_Type;
        Found : Boolean;           -- true signals EOF
        Error_Message : Display_Line(Error_Line); -- displayed no ship
        Data_Message : Display_Line(Data_Line); -- if shipments
        Total_Weight_Temp : Weight_Type;
    begin
        Detail.Open(Pno);
        Detail.Fetch(Tuple, Found); -- get first line of result
        if not Found then -- no such part
            Display_The_Line_B(Error_Message);
        end if;
        while Found loop
            Data_Message.Pno := SQL_Char_Not_Null(Tuple.Pno);

            if Is_Null(Tuple.Sname) then
                Data_Message.Sname(Null_Sname'Range) := Null_Sname;
                Data_Message.Sname_Length := Null_Sname'Length;
            else
                Data_Message.Sname_Length := Unpadded_Length(Tuple.Sname);
            end if;
        end loop;
    end;
end Example_B;

```

```

        Data_Message.Sname(Data_Message.Sname'First ..
        Data_Message.Sname'First + Data_Message.Sname_Length - 1)
    :=
        SQL_Char_Not_Null(Without_Null_Unpadded(Tuple.Sname));
    end if;

-- An example of mixed mode arithmetic
    assign(Total_Weight_Temp,
           Tuple.Weight * Weight_Type(Tuple.Qty));

    if Is_Null(Total_Weight_Temp) then
        Data_Message.Total_Weight(Null_Weight'Range) := Null_Weight;
        Data_Message.Total_Weight_Length := Null_Weight_Length;
    else
        Data_Message.Total_Weight_Length :=
            Image(Total_Weight_Temp).Length;
        Data_Message.Total_Weight(Data_Message.Total_Weight'First ..
            Data_Message.Total_Weight'First +
            Data_Message.Total_Weight_Length - 1)
        :=
            Without_Null(SQL_Char_Type(Image(Total_Weight_Temp)));
    end if;
    Display_The_Line_B(Data_Message);      -- display this line
    Detail.Fetch(Tuple, Found);           -- get next line
end loop;
Detail.Close;
end;
end Example_B;

```

Figure 8-10: Example_B


```

with Conversions, SQL_Standard, SQL_Communications_Pkg,
    SQL_Database_Error_Pkg, Example_Concrete_Module;
use Conversions, SQL_Standard, SQL_Communications_Pkg,
    SQL_Database_Error_Pkg;
package body Example_B_Module is

    package Conc renames Example_Concrete_Module;

    use Weight_Ops, QTY_Ops;

    package body Detail is

        procedure Open (Pno : in Pno_Not_Null) is

            begin
                Conc.DetailOpen(Char(Pno), SQLCODE);
                if SQLCODE /= 0 then
                    Process_Database_Error;
                    raise SQL_Database_Error;
                end if;
            end Open;

        procedure Fetch (Tuple : in out Detail_Record_Type;
            Found : out Boolean) is

            Sname : Char(Sname_Not_Null'Range);
            Weight, Qty : Int;
            Sname_Indic, Weight_Indic, Qty_Indic : Indicator_Type;
            begin
                Conc.FetchDetail(Char(Tuple.Pno),
                    Sname, Sname_Indic,
                    Qty, Qty_Indic,
                    Weight, Weight_Indic,
                    SQLCODE);

                if SQLCODE in Not_Found then          -- end of file
                    Found := False;
                elsif SQLCODE in SQL_Error then -- unrecoverable error
                    Process_Database_Error;
                    raise SQL_Database_Error;
                else                                  -- a tuple is returned
                    assign(tuple.Sname,
                        SNAME_Base(Convert(Sname, Sname_Indic)));
                    assign(tuple.Qty,
                        QTY_Type(Convert(Qty, Qty_Indic)));
                    assign(tuple.Weight,
                        Weight_Type(Convert(Weight, Weight_Indic)));
                    Found := true;
                end if;
            end Fetch;

        procedure Close is
            begin
                Conc.CloseDetail(SQLCODE);
                if SQLCODE in SQL_Error then
                    Process_Database_Error;
                    raise SQL_Database_Error;
                end if;
            end Close;

        end Detail;
    end Example_B_Module;

```

Figure 8-11: The Abstract Module Body for Example_B

Example_C

Example_C illustrates a database update. The user enters a supplier number and a signed integer. If a supplier with that number exists in the database, and if that supplier's status is not null, the integer is added to the supplier's status. If the supplier's status is null, it is replaced by the value of the integer. In other words, for this update, the null value is treated as though it were zero.

The SQL statements for Example_C appear in Figure 8-12 and the abstract module specification in Figure 8-13. In the current SQL standard, two SQL update statements are needed. One statement is used for the case that the original status is null; the other statement is used in the remaining case. (In the SQL2 standard, this update can be performed by a single statement.) Hence, it becomes essential that the application first read the relevant supplier data to determine which case applies. Thus Example_C requires three SQL statements. (Since it is necessary to read the initial status, it is possible, and simpler, to calculate the updated status value in the Ada application. This would eliminate the need for one of the two update procedures, the procedure `IncrStatus`. An attempt to set status to an invalid value, one not in the range of the Status domain, would then be trapped in the Ada application. Example_C has been designed so that the DBMS will trap illegal updates, in order to illustrate a method by which the SAME can handle that phenomenon.) The text of Example_C is found in Figure 8-14.

A new abstract domain, `Increment`, has been defined for this example. This domain does not describe any database data, but it does describe data passed across the abstract interface. (The package `Increment_Definition_Pkg` is given in Figure 8-15.) The new domain has been placed in a domain package by itself. It could have been placed in a domain package with other domains, had there been any reason to do so.

Although only the `_Not_Null` type within the domain definition is used, the domain is fully defined, with a null bearing type and a generic subpackage instantiation. There is some concrete benefit from that. The designer may be certain that there will never be a need for a null `Increment`, but such certainties are notoriously fallible. More importantly, for uniformity, consistency, and clarity, all data crossing the abstract interface must be of a type defined within an abstract domain in an abstract domain package. There is no time penalty for doing this, but there is a space penalty. If indeed there are never any null `Increments`, then the space occupied by the generic subpackage is wasted. (Some compilers may be intelligent enough to recover the wasted space.) If the space is available, the benefits of uniformity are worth the price.

The `AcquireSupplier` procedure returns an entire S tuple, even though, apparently, only the status value is of interest. This is acceptable, although it may negatively affect performance. This may be an artifact of reuse. It is likely that a software development organization writing database applications will develop procedures for accessing single tuples by key. Such procedures can be reused, as may be the case here.

The abstract procedures representing the two SQL UPDATE statements have an attached result parameter that has a locally defined enumeration type. As can be seen, these procedures can terminate in four possible ways: successfully, indicating that the requested update occurred; with a constraint violation, indicating that the update did not occur due to the new status's being out of range; with a permission violation, indicating the user does not have permission to update supplier statuses; and with no record found. The last condition is

a logical impossibility, since the update is preceded by an acquisition of the record to be updated. It may be argued that this condition should not be returned to the application, but rather trigger the standard error-processing path, as it indicates some unrecoverable error.

The Boolean-valued function `Choose` filters the suppliers based on a static property contained in the function body. This function is admittedly a contrivance designed to illustrate aspects of the SAME's logical processing. Its discussion is delayed until after the discussion of the abstract module body for `Example_C`. That code can be found in Figure 8-16.

The two update procedure bodies in Figure 8-16 are essentially identical, differing only in the concrete procedure which they call. Their function is to analyze the `SQLCODE` value returned in one of the four allowable cases. Constraint and permission violations are not thoroughly covered by the current SQL standard. That standard describes user authorizations, but does not describe the result of an authorization violation. The current standard does not cover data integrity constraints at all, although most SQL DBMSs do. Thus, the `SQLCODE` values to be looked for are dependent upon the DBMS in use. The code in Figure 8-16 is designed for use with RTI's Ingres DBMS. If this code were to be ported to a different DBMS, the constants `Constraint_Violation` and `Permission_Violation` would have to be redefined. (Notice that to Ingres, a constraint violation is signalled as a no-record-found condition. The abstract module code has been deliberately written to check for `Constraint_Violation` first. Had this code been written for some other DBMS and ported to Ingres, some recoding might have been necessary.)

```

PROCEDURE AcquireSupplier
  Sno_In Char (5)
  Sno_Out Char (5)
  Sname Char Sname_Indic Smallint
  Status Int Status_Indic Smallint
  City Char City_Indic Smallint
  SQLCODE;

  SELECT Sno, Sname, Status, City
  INTO Sno_Out,
        Sname INDICATOR Sname_Indic,
        Status INDICATOR Status_Indic,
        City INDICATOR City_Indic
  FROM S
  WHERE Sno = Sno_In;

PROCEDURE IncrStatus
  Increment Int
  Sno_In Char (5)
  SQLCODE;

  UPDATE S
  SET Status = Status + Increment
  WHERE S.Sno = Sno_In;

PROCEDURE SetStatus
  Increment Int
  Sno_In Char (5)
  SQLCODE;

  UPDATE S
  SET Status = Increment
  WHERE S.Sno = Sno_In;

```

Figure 8-12: The SQL Procedures for Example_C

```

with Increment_Definition_Pkg, Suppliers_Definition_Pkg,
    City_Definition_Pkg;
Use Increment_Definition_Pkg, Suppliers_Definition_Pkg,
    City_Definition_Pkg;
package Example_C_Module is

    type Supplier_Record_Type is record
        Sno : SNO Not Null;
        SName : SNAME_Type;
        Status : Status_Type;
        City : City_Type;
    end record;

    type Update_Status_Result_Type is (Success,
                                       No_Supplier,
                                       Constraint_Violated,
                                       Permission_Denied);

    procedure AcquireSupplier (Sno_In : in Sno_Not_Null;
                              Supplier_Record : in out Supplier_Record_Type;
                              Found : out Boolean);

    procedure IncrStatus (Sno : in Sno_Not_Null;
                        Increment : in Status_Increment_Not_Null;
                        Result : out Update_Status_Result_Type);

        -- adds Increment (signed quantity) to Status
        -- of Supplier Sno. Result is Constraint_Violated if
        -- updated status violates constraint on Status. Result is
        -- No_Supplier if Sno is not in database or its Status
        -- is Null. Result is Success if the Supplier
        -- with number Sno has had
        -- his Status incremented by the value of Increment

    procedure SetStatus (Sno : in Sno_Not_Null;
                        Increment : in Status_Increment_Not_Null;
                        Result : out Update_Status_Result_Type);

        -- Sets Status of Supplier Sno to Increment
        -- Result is Constraint_Violated if updated Status
        -- violates constraint (e.g., is negative).
        -- Result is No_Supplier if Sno is not in database
        -- or its Status is not Null. Result is Success if
        -- the Supplier with number Sno has had his Status
        -- set to the value of Increment.

end Example_C_Module;

```

Figure 8-13: The Abstract Module for Example_C

```

with Suppliers_Definition_Pkg, Parts_Definition_Pkg,
    QTY_Definition_Pkg, Increment_Definition_Pkg, Example_C_Module,
    SQL_Base_Types_Pkg;
use Suppliers_Definition_Pkg, Parts_Definition_Pkg,
    QTY_Definition_Pkg, Increment_Definition_Pkg, Example_C_Module,
    SQL_Base_Types_Pkg;

separate (Driver)
procedure Example_C (Sno : Sno_Not_Null;
    Increment : Status_Increment_Not_Null) is

    -- A filter on suppliers. Serves to illustrate SAME logic.
    function Choose (A_Supplier : Supplier_Record_Type)
        return boolean is separate;

    -- The display procedure will not be shown
    procedure Display_The_Line_C (Message : SQL_Char_Not_Null)
        is separate;

begin
    declare
        -- Messages to be displayed to user indicating status of update
        No_Supplier_Msg : constant SQL_Char_Not_Null :=
            "The Supplier " & SQL_Char_Not_Null(Sno)
            & " Does Not Exist in the Database";
        Constraint_Violation : constant SQL_Char_Not_Null :=
            "Your attempted status modification " &
            "violates database constraints";
        Update_Successful : constant SQL_Char_Not_Null :=
            "Status successfully updated";
        Not_Chosen : constant SQL_Char_Not_Null :=
            "You may not Update the Status of Supplier " &
            SQL_Char_Not_Null(Sno);
        Permission_Denial : constant SQL_Char_Not_Null :=
            "You do not have permission to update Supplier data";
        Unknown_Error : constant SQL_Char_Not_Null :=
            "The Supplier " & SQL_Char_Not_Null(Sno) &
            "has inexplicably disappeared from the database." &
            " Contact a service representative.";
        -- objects for concrete module communication
        Supplier : Supplier_Record_Type;
        Exists : Boolean;
        Results : Update_Status_Result_Type;
    begin
        AcquireSupplier(Sno, Supplier, Exists); -- get initial status
        if not Exists then
            Display_The_Line_C(No_Supplier_Msg); -- no such supplier
        elsif Choose(Supplier) then -- filter suppliers
            if Is_Null(Supplier.Status) then -- decide which SQL statement
                SetStatus(Sno, Increment, Results); -- to call
            else
                IncrStatus(Sno, Increment, Results);
            end if;
            case Results is -- tell user status of update
                when No_Supplier =>
                    Display_The_Line_C(Unknown_Error);
                when Constraint_Violated =>
                    Display_The_Line_C(Constraint_Violation);
                when Permission_Denied =>
                    Display_The_Line_C(Permission_Denial);
                when Success =>
                    Display_The_Line_C(Update_Successful);
            end case;
        end if;
    end;
end;

```

```

        else
            Display_The_Line_C(Not_Chosen);    -- status when filtered out
        end if;
    end;
end Example_C;

```

Figure 8-14: Example_C

```

with Suppliers_Definition_Pkg, SQL_Int_Pkg;
use Suppliers_Definition_Pkg, SQL_Int_Pkg;
package Increment_Definition_Pkg is

    type Status_Increment_Not_Null is new SQL_Int_Not_Null
        range -SQL_Int_Not_Null(Status_Not_Null'Last) ..
            SQL_Int_Not_Null(Status_Not_Null'Last);
    type Status_Increment_Type is new SQL_Int;
    package Status_Increment_Ops is new
        SQL_Int_Ops(Status_Increment_Type, Status_Increment_Not_Null);

end Increment_Definition_Pkg;

```

Figure 8-15: The Package Increment_Definition_Pkg

```

with Conversions, SQL_Standard, SQL_Communications_Pkg,
     SQL_Database_Error_Pkg, Example_Concrete_Module;
use Conversions, SQL_Standard, SQL_Communications_Pkg,
     SQL_Database_Error_Pkg;
package body Example_C_Module is

    package Conc renames Example_Concrete_Module;

    use SNAME_Ops, Status_Ops, City_Ops;

    Constraint_Violation : constant := 100; -- implementation defined
                                         -- the value of SQLCODE
                                         -- when an update would violate
                                         -- a constraint
    Permission_Violation : constant := -1; -- implementation defined
                                         -- the value of SQLCODE
                                         -- when a user does not have
                                         -- update permission

    procedure AcquireSupplier (Sno_In : in Sno_Not_Null;
                              Supplier_Record : in out Supplier_Record_Type;
                              Found : out Boolean) is

        Sname_c : Char(Sname_Not_Null'Range);
        Status_c : Int;
        City_c : Char(City_Not_Null'Range);
        Sname_Indic, Status_Indic, City_Indic : Indicator_Type;
    begin
        Conc.AcquireSupplier(Char(Sno_In),
                             Char(Supplier_Record.Sno),
                             Sname_c, Sname_Indic,
                             Status_c, Status_Indic,
                             City_c, City_Indic,
                             SQLCODE);

        If SQLCODE in Not_Found then
            Found := False;
        elsif SQLCODE /= 0 then
            Process_Database_Error;
            raise SQL_Database_Error;
        else
            Found := True;
            assign(Supplier_Record.Sname,
                  SNAME_Base(Convert(Sname_c, Sname_Indic)));
            assign(Supplier_Record.Status,
                  Status_Type(Convert(Status_c, Status_Indic)));
            assign(Supplier_Record.City,
                  CITY_Base(Convert(City_c, City_Indic)));
        end If;
    end AcquireSupplier;

    procedure IncrStatus (Sno : in Sno_Not_Null;
                          Increment : in Status_Increment_Not_Null;
                          Result : out Update_Status_Result_Type) is

    begin
        Conc.IncrStatus(Int(Increment),
                       Char(Sno),
                       SQLCODE);

        If SQLCODE = Constraint_Violation then
            -- update refused: constraints

```



```

        Result := Constraint_Violated;
    elsif SQLCODE = Permission_Violation then
        -- update refused: permission
        Result := Permission_Denied;
    elsif SQLCODE in Not_Found then
        Result := No_Supplier;           -- Sno not in database
                                         -- or Status Null
    elsif SQLCODE /= 0 then             -- unrecoverable error
        Process_Database_Error;
        raise SQL_Database_Error;
    else
        Result := Success;              -- successful completion
    end If;
end IncrStatus;

procedure SetStatus (Sno : in Sno_Not_Null;
                    Increment : in Status_Increment_Not_Null;
                    Result : out Update_Status_Result_Type) is
    -- This logic is identical to IncrStatus except
    -- concrete procedure SetStatus is called
begin
    Conc.SetStatus(Int(Increment),
                  Char(Sno),
                  SQLCODE);

    If SQLCODE = Constraint_Violation then
        -- update refused: constraints
        Result := Constraint_Violated;
    elsif SQLCODE = Permission_Violation then
        -- update refused: permission
        Result := Permission_Denied;
    elsif SQLCODE in Not_Found then
        Result := No_Supplier;           -- Sno not in database
    elsif SQLCODE /= 0 then             -- unrecoverable error
        Process_Database_Error;
        raise SQL_Database_Error;
    else
        Result := Success;              -- successful completion
    end If;
end SetStatus;
end Example_C_Module;

```

Figure 8-16: The Abstract Module Body for Example_C

```

with City_Definition_Pkg; use City_Definition_Pkg;
separate (Driver.Example_C)
function Choose (A_Supplier : Supplier_Record_Type) return boolean is

    use City_Ops;
    use Character_Set;

begin
    -- this version rejects any supplier known to be in Pittsburgh
    if A_Supplier.City = With_Null("Pittsburgh") then
        return false;
    else
        return true;
    end if;
end Choose;

```

Figure 8-17: Choose - Version 1

```

with City_Definition_Pkg, SQL_Boolean_Pkg;
use City_Definition_Pkg, SQL_Boolean_Pkg;
separate (Driver.Example_C)
function Choose (A_Supplier : Supplier_Record_Type) return boolean is

    use City_Ops;
    use Character_Set;

begin
    -- this version rejects any supplier that might be in Pittsburgh

    case Equals(A_Supplier.City, With_Null("Pittsburgh")) is
        when True | Unknown =>
            return false;
        when False =>
            return true;
    end case;
end Choose;

```

Figure 8-18: Choose - Version 2

Illustrations of Three-Valued Logic

This section concludes with a discussion of the Choose function in Example_C. As mentioned, this function has been contrived for the purpose of illustrating logical processing within the SAME. Five separate versions of Choose, illustrating different aspects of that processing, will be presented. The first two versions appear in Figures 8-17 and 8-18. These two versions are both concerned with the city of Pittsburgh. In the first version, the function returns *false* for any supplier whose city value is Pittsburgh. The second version returns *false* for suppliers whose city is either unknown (null) or Pittsburgh. This version needs visibility to SQL_Boolean_Pkg in order to have the enumeration literals in the case alternatives correctly identified. The first version deals only with known information, with the ability to establish a fact; the second version deals with uncertainty, with the inability to disprove a fact. In other words, the version in Figure 8-17 looks for suppliers whose city is definitely Pittsburgh, the so-called *minimal* result; whereas, the version in Figure 8-18 looks for suppliers whose city may be Pittsburgh, the so-called *maximal* result.

In the third and fourth versions of Choose, displayed in Figures 8-19 and 8-20, suppliers are selected based on their status values. The third version, in Figure 8-19, resembles the first version, in Figure 8-17, in that it rejects only those suppliers whose status is known to not exceed the specified value. Similarly, the fourth (Figure 8-20) resembles the second (Figure 8-18), in rejecting those suppliers whose status values *might not* exceed the given value. The fourth version works by the double negation principle; suppliers are rejected if it is not known that their status values exceed the given value. The fourth version could have been coded in the style of the second version, using a case statement whose alternatives are guarded by literals of the Boolean_with_Unknown enumeration type. However, the second example (Figure 8-18) cannot be coded in the style of the fourth, since Ada will not allow explicit overloadings of the negation of the equality operator.

The final version of Choose, shown in Figure 8-21, exemplifies mixed mode comparisons for string based values and the substring operation. This version rejects suppliers whose name contains their city as a substring. Only the definite information version is shown. Points to be noticed about Figure 8-21 are:

- The search excludes the sequence of trailing blanks in the supplier's name field.
- The search avoids the exception constraint_error in the Substring function. This and the previous point explain the upper bound on the for loop.
- The search does not require the string of trailing blanks, if any, in the city field to be present in the name field. This explains the length parameter in the Substring function call.
- It is not necessary to actually remove the trailing blanks from the City field.
- For the comparison to be syntactically valid, one of its operands must be converted to the other's type. The city operand is converted to the type of suppliers' names. The unconstrained type, SNAME_Base, is used. Were the constrained type, SNAME_Type, used here, a constraint_error would be raised due to the conflict in discriminant values, i. e., string lengths.

```

separate (Driver.Example_C)
function Choose (A_Supplier : Supplier_Record_Type) return boolean is

    use Status_Ops;
    use Character_Set;

begin
    -- this version rejects any supplier
    -- whose status is known to be less than or equal to 20

    if A_Supplier.Status <= With_Null(20) then
        return false;
    else
        return true;
    end if;
end Choose;

```

Figure 8-19: Choose - Version 3

```

separate (Driver.Example_C)
function Choose (A_Supplier : Supplier_Record_Type) return boolean is

    use Status_Ops;
    use Character_Set;

begin
    -- this version rejects any supplier
    -- whose status is not known to be greater than 20

    if not (A_Supplier.Status > With_Null(20)) then
        return false;
    else
        return true;
    end if;
end Choose;

```

Figure 8-20: Choose - Version 4

```

with City_Definition_Pkg, Suppliers_Definition_Pkg;
use City_Definition_Pkg, Suppliers_Definition_Pkg;
separate (Driver.Example_C)
function Choose (A_Supplier : Supplier_Record_Type) return boolean is

    use City_Ops, SNAME_Ops;

begin
    -- this version rejects any supplier whose name contains
    -- its city as a substring
    for i in
        1 ..
            (Unpadded_Length(A_Supplier.Sname) -
             Unpadded_Length(A_Supplier.City) - 1)
    loop
        if Substring(A_Supplier.Sname, i, Unpadded_Length(A_Supplier.City))
           =
           SNAME_Base(A_Supplier.City) then
            return False;
        end if;
    end loop;
    return True;
end Choose;

```

Figure 8-21: Choose - Version 5

9. Advanced DBMS Applications

This chapter deals with specialized applications of SQL DBMS technology; in particular, with applications that require dynamic SQL services and those Ada DBMS applications which use Ada tasking. It should be noted that ANSI standard SQL [2] supports neither of these features.³¹ There are DBMS implementations on the market which provide support for one or both of these facilities. The discussion in this section cannot take the details of these implementations into account. The reader will need to adapt the methods of this section to the target DBMS.

A second note of caution must be introduced into this section. Whereas the ideas in other sections of these guidelines have been verified, and all of the code has been compiled and executed, the author did not have at his disposal a DBMS which supported either of the classes of applications discussed in this section. Therefore, the code presented here has not been executed, although it has been compiled, and the ideas have not been directly tested against any DBMS.

9.1. Dynamic SQL

As has been shown in previous sections, SQL statements can take runtime parameters. This parameterization is limited to those parts of an SQL statement in which a constant may appear. In the examples of Chapter 8, SQL statements were parameterized with Supplier and Part numbers. If a needed DBMS service is to be parameterized by something other than a constant, this can be done with dynamic SQL. If, for example, an update application allows for the modification of various sets of dynamically specified columns using various sets of dynamically specified update expressions and various sets of dynamically specified search conditions, it may choose to use dynamic SQL. If the amount of variation is very small, it may be preferable for the application designer to produce a small set of static SQL update statements and choose the statement to execute at runtime. Dynamic SQL applications are harder to write than static SQL applications and add runtime overhead. A good heuristic to follow is to avoid the use of dynamic SQL whenever feasible.

A full description of dynamic SQL is inappropriate for these guidelines. There follows a brief description of dynamic SQL based on the proposals in [3].

The SQL statement to be dynamically executed is created by the application as a character string. This string is presented to the DBMS as the operand of a PREPARE statement. If the statement is not a SELECT statement, i.e., if it is an INSERT, UPDATE, DELETE or one of a handful of other, bookkeeping statements (see [3]), it may then be EXECUTED. A cursor must be declared for SELECT statements. Once declared, the cursor is OPENed, FETCHed and CLOSED as in static SQL. Thus, the mental model of dynamic SQL operation is very much the same as for static SQL.

Dynamic SQL applications can be placed along a continuum whose end points may be called "fully dynamic" and "slightly dynamic." Fully dynamic applications are generalized system software utilities. They typically provide an ad hoc browsing and updating capability.

³¹The follow-on ANSI standard [3] has support for dynamic SQL.

(Most SQL DBMS offer an interactive version of SQL. However, SQL is probably not a good end user language.) These applications are often supplied by the DBMS vendor or by third parties and are written with no knowledge of the schema of the database against which they execute. Slightly dynamic applications offer more restricted services to their users. They are written with full knowledge of the target database schema, its semantics, and the abstract domains involved.

The central distinction between static and dynamic SQL statement execution is the manner in which runtime parameters are passed. SQL2 offers three distinct methods of passing parameters to dynamically prepared statements. Each dynamic statement³² has a USING clause whose operand specifies the manner in which parameters are being passed. In the simplest case, this operand is a list of identifiers. This alternative can and should be used whenever the number and type of the parameters of the statement, i.e., its parameter profile, do not vary and the dynamically varying parts of the statement lie elsewhere (e.g., in the use of these parameters in a search condition). Such applications lie at the slightly dynamic end of the continuum. When the list of identifiers option of the USING clause appears, the abstract procedure declaration corresponding to the dynamic statement is identical to its static counterpart in its use of row records, abstract domain types, and result parameters.³³

Example

Suppose a program wishes to execute an UPDATE statement which always takes a part number, a color, and a weight, sometimes updating the color and sometimes updating the weight. Assuming this is to be done with dynamic SQL, two dynamic statements are needed: PREPARE and EXECUTE. (In practice, an EXECUTE IMMEDIATE, which performs both functions, could be used. Two statements are used here for purposes of illustration.) The module procedures are:

```
PROCEDURE STMT_PREP
  STMT_TO_PREP CHAR(100)
  SQLCODE;

  PREPARE ST FROM STMT_TO_PREP;

PROCEDURE UPDATE_EXEC
  PNO CHAR (5)
  WEIGHT INT WEIGHT_INDIC SMALLINT
  COLOR CHAR (6) COLOR_INDIC SMALLINT

  EXEC ST USING PNO,
              WEIGHT_INDICATOR WEIGHT_INDIC,
              COLOR_INDICATOR COLOR_INDIC;
```

³²Dynamic SQL statements, e.g., PREPARE, EXECUTE, dynamic OPEN, and dynamic FETCH, are distinct from dynamically prepared SQL statements, e. g., SELECT, UPDATE, INSERT. The dynamic SQL statements are those which are executed by a dynamic SQL program to accomplish the database operations specified by the dynamically prepared SQL statements.

³³SQL2's notions of extended statement identifier and extended cursor name, to be described, are runtime parameters which may be needed at the abstract interface, even in this case.

The abstract module procedure declarations corresponding to these module procedure are:

```
procedure Stmt_Prep (Stmt_To_Prep : in SQL_Char_Not_Null);

procedure Update_Exec (Pno : in Pno_Not_Null;
                      Weight : in Weight_Type;
                      Color : in Color_Type);
```

Result parameters can, of course, be attached to either or both of these procedures.

Applications which require SQL statements whose parameter profiles vary dynamically must be "polymorphic," that is, able to deal with a variety of types at runtime. Although Ada is not a polymorphic programming language, the Ada variant record construct can be used to simulate polymorphism, provided that the set of possible runtime types is known at compile time. Furthermore, each variant will typically require path segments unique to it. It is best if the number of types is kept small.

SQL2 offers two methods of passing parameters to dynamically prepared statements whose parameter profiles vary dynamically. Both methods are based on the *<dynamic using descriptor area structure>* or SQLDA. In the first of the two methods the SQLDA is allocated by the application program and exists in its name space. In the second method, the SQLDA is allocated by the DBMS and exists in its name space. In Ada terms, the distinction is that between visible and private declarations of the SQLDA type. The first of these alternatives, the visible SQLDA, will be described first, as the second alternative, the functional approach, is defined in terms of it.

The definition of the SQLDA structure in PL/I can be found in Figure 9-1. The ANSI proposal does not allow this structure in Ada. This is subject to change before the standard is approved and, of course, there are no implementations conformant with the SQL2 proposal. As was mentioned, the reader will need to adapt the discussion in this section to the target DBMS in any case. A proposed definition for an SQLDA in Ada appears in Figure 9-2. The package SQL_Standard_Dynamic is like the package SQL_Standard in that it describes data crossing the concrete interface.

```
DCL 1 SQLDA
  2 SQLN BIN FIXED,
    /* max nbr of parameters*/
  2 SQLD BIN FIXED,
    /* actual nbr of parameters */
  2 SQLVAR (SQLSIZE REFER (SQLN)),
    3 SQLDATA PTR,
    /* points to the data */
    3 SQLIND PTR,
    /* points to the indicator parm */
  3 SQLTYPE BIN FIXED,
    /*integer encode of type */
  3 SQLNULLABLE BIN FIXED,
    /* is there an indicator parm? */
  3 SQLLEN BIN FIXED,
    /* character length, numeric precision */
  3 SQLSCALE BIN FIXED,
    /* numeric scale */
  3 SQLNAME CHAR (k) VAR;
    /* column name if applicable */
DCL SQLSIZE BIN FIXED;
```

Figure 9-1: SQLDA in PL/I

The implementation-specific type `SQL_Dynamic_Datatypes_Base` is used to choose the appropriate integer type as defined by the DBMS. The constants of this type, `Dynamic_Char`, etc., define the integer encoding of types as specified by ANSI [3]. The constant `Not_Specified` is used as the default for the discriminant of the `SQL_Dynamic_Parameter` type. The subtype `SQL_Dynamic_Datatypes` is used as the type of the discriminant to obviate the need for an **others** variant.

The type of the `SQLDATA` component of the `SQLVAR_Component_Type` (`SQL_Dynamic_Parameter`) is a variant record of access types. The objects accessed by these variants are of types declared in `SQL_Standard` (or the not null decimal type in `SQL_Decimal_Pkg`). The `SQLLEN` and `SQLSCALE` fields, which give length, precision, and scale information, are no longer present as fields, but are now attributes (or discriminants) of the accessed objects.

The dynamic SQL `DESCRIBE` statement takes a statement identifier and an `SQLDA` object and fills in the type information in the `SQLDA` from the prepared statement identified by the identifier. When issued in conjunction with the definitions of Figure 9-2, the `DESCRIBE` statement must also allocate the space for the values of the dynamic parameters described by each `SQLVAR_Component_Type` object, in order to return length, precision, and scale information. The values themselves may be left undefined by `DESCRIBE`. This behavior is slightly different from the behavior of `DESCRIBE` in [3], Section 12.7.

The types `Extended_Cursor_Type` and `Extended_Statement_Type` are used for the *<extended cursor name>* and *<extended statement identifier>* of SQL2 [3]. Briefly, the connection between dynamic statements operating on a dynamically prepared statement (e.g., `PREPARE` and `EXECUTE`) is via a *<statement identifier>* which may be either a constant or a variable. In the example given earlier, the token `ST` is a constant statement identifier. Similarly, the connection between dynamic open, close, and fetch and the prepared select statement on which they operate is via a *<cursor identifier>*, which may be either a constant or a variable. (When a cursor is a runtime variable, a *<dynamic declare cursor>* statement must be executed to form the connection between the prepared select statement and the cursor.) An object containing an extended statement identifier has type `Extended_Statement_Type`; an object containing a dynamic extended cursor has type `Extended_Cursor_Type`.

```

with SQL_Standard, SQL_Decimal_Pkg;
use SQL_Standard, SQL_Decimal_Pkg;
package SQL_Standard_Dynamic is

type Extended_Cursor_Type is implementation defined;
type Extended_Statement_Type is implementation defined;
type SQL_Dynamic_Datatypes_Base is range implementation defined;

Maybe_Null_Indic : constant Indicator_Type := 1;
-- value of SQLNULLABLE if nulls allowed
subtype Null_Indication is Indicator_Type range
Indicator_Type'First .. -1;
-- value of indicator if value is null

-- types to describe column names
SQL_Column_Name_Length : constant := 18; -- set in SQL2 standard
subtype SQL_Column_Name_Length_Type is
positive range 1..SQL_Column_Name_Length;
subtype SQLNAME_Type is Char(SQL_Column_Name_Length_Type);

-- These constants capture the encoding of SQL Types as integers
-- as given by SQL2.
Not_Specified : constant SQL_Dynamic_Datatypes_Base := 0;
Dynamic_Char : constant SQL_Dynamic_Datatypes_Base := 1;
Dynamic_Numeric : constant SQL_Dynamic_Datatypes_Base := 2;
Dynamic_Decimal : constant SQL_Dynamic_Datatypes_Base := 3;
Dynamic_Int : constant SQL_Dynamic_Datatypes_Base := 4;
Dynamic_Smallint : constant SQL_Dynamic_Datatypes_Base := 5;
Dynamic_Float : constant SQL_Dynamic_Datatypes_Base := 6;
Dynamic_Real : constant SQL_Dynamic_Datatypes_Base := 7;
Dynamic_Double_Precision : constant SQL_Dynamic_Datatypes_Base := 8;

subtype SQL_Dynamic_Datatypes is SQL_Dynamic_Datatypes_Base
range Not_Specified .. Dynamic_Double_Precision;

-- access types for components of SQL_Dynamic_Parameter
type Char_Access is access Char;
type Decimal_Access is access SQL_Decimal_Not_Null;
type Int_Access is access Int;
type Smallint_Access is access Smallint;
type Real_Access is access Real;
type Double_Precision_Access is access Double_Precision;
type SQL_Dynamic_Parameter (SQLTYPE : SQL_Dynamic_Datatypes:=Not_Specified)
is record
case SQLType is
when Not_Specified =>
null;
when Dynamic_Char =>
Char_Value : Char_Access;
when Dynamic_Decimal | Dynamic_Numeric =>
Decimal_Value : Decimal_Access;
when Dynamic_Int =>
Int_Value : Int_Access;
when Dynamic_Smallint =>
Smallint_Value : Smallint_Access;
when Dynamic_Real =>
Real_Value : Real_Access;
when Dynamic_Double_Precision | Dynamic_Float =>
Double_Precision_Value : Double_Precision_Access;
end case;
end record;

type SQLVAR_Component_Type is record

```

```

SQLDATA : SQL_Dynamic_Parameter;
SQLNULLABLE : Indicator_Type;
SQLIND : Indicator_Type;
SQLNAME1 : SQL_Column_Name_Length_Type;
SQLNAME : SQLNAME_Type;
end record;

type SQLVAR_Type is
  array (Int range <>) of SQLVar_Component_Type;

type SQLDA (SQLN : Int) is record
  SQLD : Int;
  SQLVAR : SQLVAR_Type (1 .. SQLN);
end record;

end SQL_Standard_Dynamic;

```

Figure 9-2: The Package SQL_Standard_Dynamic

The package `SQL_Standard_Dynamic` is like the package `SQL_Standard` in describing data at the level of the concrete interface. Before describing an abstract interface for dynamic SQL, it is first necessary to consider what the goals of an abstract interface design for dynamic SQL should be.

As mentioned earlier, fully dynamic SQL applications are general system software supporting ad hoc user interactions. As such, these programs are independent of any database schema, which is to say, of the semantics of the stored data. These programs do not deal with Part Numbers, Supplier Names, Weights, etc. They deal with character strings, integers, etc. For this reason, the suggested definition of an abstract SQLDA in Figure 9-3 does not allow for user defined types. However, fully dynamic SQL applications can be provided with the standard SAME treatment of null values and the standard SAME treatment of database exceptional conditions.

The package `SQL_Dynamic_Pkg` in Figure 9-3 presents a set of abstract types closely modeled on the set of concrete types in `SQL_Standard_Dynamic`. The underlying, "scalar" types have been changed to types suitable for an abstract interface. These types are defined in the abstract domain package, `SQL_Base_Types_Pkg`, which was introduced in Figure 3.8. The types of the objects accessed by components of `SQL_Dynamic_Parameter` in `SQL_Dynamic_Pkg` are all of null bearing types. It is possible to introduce the non-null bearing types, or a set of abstract types, into this list of components, but at the expense of increased application complexity. Each variant of `SQL_Dynamic_Parameter` will require an execution path segment of its own. There is good reason to keep the number of such variants small.

```

with SQL_Base_Types_Pkg, SQL_Standard_Dynamic;
use SQL_Base_Types_Pkg;
package SQL_Dynamic_Pkg is

-- These next definitions deal with names of columns
subtype SQL_Column_Name_Length_Type is
    positive range 1..SQL_Standard_Dynamic.SQL_Column_Name_Length;
subtype SQLNAME_Type is SQL_Char_Not_Null(SQL_Column_Name_Length_Type);

-- The discriminant is now an enumeration type
type SQL_Dynamic_Datatypes is
    (Not_Specified,
     Dynamic_Char, Dynamic_Decimal,
     Dynamic_Int, Dynamic_Smallint,
     Dynamic_Real, Dynamic_Double_Precision);

-- access types access null bearing types in Base_Type_Pkg
type Char_Access is access SQL_Char_Type;
type Decimal_Access is access SQL_Decimal_Type;
type Int_Access is access SQL_Int_Type;
type Smallint_Access is access SQL_Smallint_Type;
type Real_Access is access SQL_Real_Type;
type Double_Precision_Access is access SQL_Double_Precision_Type;

type SQL_Dynamic_Parameter (SQLTYPE :SQL_Dynamic_Datatypes := Not_Specified
is record
    case SQLType is
        when Not_Specified =>
            null;
        when Dynamic_Char =>
            Char_Value : Char_Access;
        when Dynamic_Decimal =>
            Decimal_Value : Decimal_Access;
        when Dynamic_Int =>
            Int_Value : Int_Access;
        when Dynamic_Smallint =>
            Smallint_Value : Smallint_Access;
        when Dynamic_Real =>
            Real_Value : Real_Access;
        when Dynamic_Double_Precision =>
            Double_Precision_Value : Double_Precision_Access;
    end case;
end record;

type SQLVAR_Component_Type is record
    SQLDATA : SQL_Dynamic_Parameter;
    SQLNAME1 : SQL_Column_Name_Length_Type;
    SQLNAME : SQLNAME_Type;
end record;

type SQLVAR_Type is
    array (SQL_Int_Not_Null range <>) of SQLVAR_Component_Type;

type SQLDA (SQLN : SQL_Int_Not_Null) is record
    SQLD : SQL_Int_Not_Null;
    SQLVAR : SQLVAR_Type (1 .. SQLN);
end record;
end SQL_Dynamic_Pkg;

```

Figure 9-3: The Package SQL_Dynamic_Pkg

With the definitions of Figures 9-3 and 9-2 at hand, it is possible to write an abstract module supporting a dynamic application. The module allocates and maintains a local object of the concrete SQLDA type, as defined by the package `SQL_Standard_Dynamic` in Figure 9-2, and exports to the application subprograms which take parameters of the abstract SQLDA type, given by Figure 9-3. The module then translates between the two formats on each subprogram call. Although such modules are possible, they may not be desirable, particularly when built for a DBMS which does not directly support either SQLDA type. (Of course, there are no DBMSs which support these types at this time.) A module which operates in this way requires an excessive amount of data movement. The information in the SQLDA would first be stored in an SQLDA structure local to the DBMS (probably in either C or PL/I, the only languages currently supporting an SQLDA in SQL2), translated to the concrete Ada SQLDA, and then translated to the abstract SQLDA. These translations are done field by field. Since the purported advantage of an SQLDA structure is runtime efficiency, the overhead of these translations is unacceptable. The remaining alternative to dynamic parameter passing, the functional approach,³⁴ eliminates much of this data translation.

The functional approach treats the SQLDA as a private type declared, from the application program's point of view, behind the abstract interface. The application program allocates objects of the SQLDA type using an SQL-defined allocation procedure whose syntax is:

```
ALLOCATE SQLDESCRIPTOR <sqlda descriptor name>
    WITH MAX <occurrences>
```

where *<sqlda descriptor name>* is a character string parameter and *<occurrences>* is an integer parameter. This statement appears at the abstract interface as the following procedure declaration:³⁵

```
procedure Allocate (SQLDA_Name : SQL_Char_Not_Null;
                   Max : SQL_Int_Not_Null);
```

A call to this procedure having the form:

```
Allocate (SQLDA_Name => "SQLDA_Object",
         Max          => 10);
```

creates an SQLDA structure with 10 occurrences of the SQLVAR component (i.e., an SQLN value of 10). This structure can be referenced by the name "SQLDA_Object" as in the procedure call:

```
Deallocate (SQLDA_Name => "SQLDA_Object");
```

which calls a procedure defined by the SQL syntax:

```
DEALLOCATE SQLDESCRIPTOR <sqlda descriptor name>
```

There is no need for more than one Allocate or Deallocate statement in any module.

The type information within an SQLDA is supplied as the result of a DESCRIBE (or DESCRIBE INPUT) statement. These statements take a prepared statement identifier and an SQLDA object name. (This information can also be modified, to within implementation-defined limits,

³⁴The functional approach does not appear in [3]. It is contained in an accepted change to SQL2, which can be found in [10]. The ensuing discussion is based on [10], which may differ from the description of the functional approach that will appear in the final standard. The differences should be minor and should not affect an abstract interface providing a functional approach to an Ada application.

³⁵The Ada code in these following examples uses types in `SQL_Base_Types_Pkg`. It may be desirable to use specially designed types, declared in a package similar in purpose, but not design, to `SQL_Dynamic_Pkg`, for the parameters in these examples.

by an application, thereby effecting runtime data conversion.) Since the SQLDA is itself hidden, two functions, GET and SET, are provided to access or modify the type information and the values of the parameters. These functions have two forms which are described by the following combined syntax:

```
(GET | SET) <sqlda descriptor name>
    [VALUES <sqlvar number>] <parameter associations>
```

The *<parameter associations>* determine what information is to be extracted from (or set into) the SQLDA. The form without the VALUES *<sqlvar number>* phrase is used to access the SQLD field, which determines the actual number of parameters used by the dynamic statement. This is the only field of an SQLDA which is not a subcomponent of the SQLVAR component. The form with the VALUES phrase accesses subcomponents of the SQLVAR component with index, relative to one, of *<sqlvar number>*.

Within [10], the *<parameter associations>* are of the form *<parameter> = <identifier>* where *<identifier>* is the name of an SQLDA field as shown in the PL/I description in Figure 9-1. (When VALUES is absent, only SQLD may appear as an *<identifier>*.) Notice that the GET (SET) statement is not itself dynamically preparable; therefore calls to these statements have parameter profiles that can be determined at compile time.

Figure 9-4 contains fragments of a "fully dynamic" Ada application using the functional interface. The example is based on [10]. The application is fully dynamic in that it uses the data types in SQL_Base_Types_Pkg.

The abstract module used by the program in Figure 9-4 contains the procedure declarations for the SQL statements which implement the functional approach to dynamic parameter passing. It is not essential that the concrete interface used by the abstract module also implement the functional approach; an SQLDA-based concrete interface is permissible. The decision can be made on performance grounds alone. The abstract module retains responsibility for null value encapsulation and SQLCODE processing. (SQLCODE processing is not explicitly used in Figure 9-4, in order to control its size. Comments indicate what might be done in a realistic setting.) The procedure Set_SQLDATA (Get_SQLDATA) gives values to (accepts values from) the DBMS. These procedures have overloaded declarations in the abstract module, one declaration for each of the null bearing types in Weak_Types_Pkg. The abstract module procedure bodies are responsible for processing the null value. For example, the body of a Set_SQLDATA procedure might be:

```
if Is_Null (SQLDATA) then
    Conc.Set_SQLNull (SQLVAR_Nbr => SQLVAR_Nbr,
                     SQLDA_Name => SQLDA_Name,
                     SQLNULLABLE => Maybe_Null_Indic,
                     SQLIND => Null_Indication_Last);
else
    Conc.Set_SQLDATA (SQLVAR_Nbr => SQLVAR_Nbr,
                     SQLDA_Name => SQLDA_Name,
                     SQLIND => 0,
                     SQLDATA => SQLDATA);
end if;
```

Similarly, the Get_SQLDATA procedure needs a concrete Get_SQLNull procedure to determine if an output value is null. These are examples of concrete procedures which do not appear at the abstract interface. Generally, that is to say, in static SQL applications, there are no such procedures. (**Note:** In the above if statement, the object Maybe_Null_Indic and the subtype Null_Indication are as defined in the package SQL_Standard_Dynamic shown in Figure 9-2.)

It is possible to envision an abstract module and application program which are less fully dynamic and use abstract domains for parameter values. Dynamic SQL requires the database to access its data dictionary at runtime. This processing could be extended to access an Ada data dictionary as well.³⁶ This would allow the application program access to the abstract domain of the parameters. However, such access would increase the complexity of the application and the runtime overhead of the abstract module. It is unclear whether the benefits of abstract typing outweigh the costs, for dynamic applications. (Note: If the abstract domain definitions are used to constrain, via range constraints, database objects in a manner which is not also supported by the DBMS, then fully dynamic update programs which do not use the abstract domain definitions may violate database constraints.)

³⁶As mentioned in the introductory chapter, the SAME - Design Committee is working on a language for automation of SAME application development. The processor for this language, whatever its final form, will certainly need an Ada data dictionary.

```

Max_SQLVAR : constant := 10; -- this limit on SQLVAR occurrences is
                           -- a property of the application and of
                           -- the DBMS implementation

Input_SQLDA : constant SQL_Char_Not_Null := "Input_SQLDA";
Output_SQLDA : constant SQL_Char_Not_Null := "Output_SQLDA";

SQLTYPE : SQL_Dynamic_Datatypes; -- type declared in SQL_Dynamic_Pkg

SQLD_Out, SQLD_In : SQL_Int_Not_Null;
Is_Fetched : boolean; -- result parameter for fetch

begin
  -- assume the dynamic statement is available in object STMT,
  -- of type SQL_Char_Not_Null. Assume also it is the only statement
  -- which will be in use at any one time. This allows for constant
  -- statement identifiers and cursor names.
  Prepare(STMT);
  -- a failure here is probably a badly formed statement. This can
  -- be trapped here, using an SQLCODE result mapping and parameter.
  Allocate(SQLDA_Name => Input_SQLDA, Max => Max_SQLVAR);
  Allocate(SQLDA_Name => Output_SQLDA, Max => Max_SQLVAR);
  -- Failure here is irrecoverable.
  Describe_In(Input_SQLDA); -- Inputs to the prepared Statement
  Describe(Output_SQLDA); -- Outputs. The statement identifier
                           -- is statically known to the module.
  -- Failure here is irrecoverable.
  Get_SQLD(SQLDA_Name => Input_SQLDA, SQLD => SQLD_In);
  -- Failure here is irrecoverable.
  if SQLD_In > 0 then
    for i in 1 .. SQLD_In loop
      Get_SQLTYPE(SQLVAR_Nbr => i,
                  SQLDA_Name => Input_SQLDA,
                  SQLTYPE => SQLTYPE);
      -- Failure here is irrecoverable.
      case SQLTYPE is
        when Dynamic_Char =>
          -- get the character string from the user.
          -- assume it is in an object called Char_Obj of type
          -- SQL_Char_Type in SQL_Base_Types_Pkg.
          Set_SQLDATA(SQLVAR_Nbr => i,
                      SQLDA_Name => Input_SQLDA,
                      SQLDATA => Char_Obj);
          -- Include an alternative
          -- for each element of SQL_Dynamic_Datatypes.
          -- The object containing the input value will be distinct
          -- in each alternative, as it will have a distinct type.
          . . .
        end case;
      end loop;
    end if;
    Get_SQLD(SQLDA_Name => Output_SQLDA, SQLD => SQLD_Out);
    if SQLD_Out = 0 then -- if no outputs, not a select
      Execute(SQLDA_Name => Input_SQLDA);
      -- There are many non successful statuses which might be
      -- trapped here: permission or constraint violation,
      -- record not found, etc. This is omitted here, as it has been
      -- fully illustrated elsewhere.
    else -- if it does have outputs, it is a select
      -- cursor does not need to be declared, as both cursor name
      -- and statement identifier are statically known to the module
      Open_Cursor(SQLDA_Name => Input_SQLDA);
      -- Failures on Open are irrecoverable.
    end if;
  end if;
end;

```

```

Fetch(SQLDA_Name => Output_SQLDA, Result => Is_Fetched);
if not Is_Fetched then
    -- perform 'no records were retrieved' processing
else
    while Is_Fetched loop
        for i in 1 .. SQLD_Out loop
            Get_SQLTYPE(SQLVAR_Nbr => i,
                SQLDA_Name => Output_SQLDA,
                SQLTYPE => SQLTYPE);
            case SQLTYPE is
                when Dynamic_Char =>
                    Get_SQLDATA(SQLVAR_Nbr => i,
                        SQLDA_Name => Input_SQLDA,
                        SQLDATA => Char_Obj);
                    -- process Char_Obj as needed
                    -- An alternative is needed
                    -- for for each type in SQL_Dynamic_Datatypes.
                    . . .
            end case;
        end loop;
        -- end of tuple processing
    end loop;
    -- end of file processing
end if;
-- end of cursor processing
Close_Cursor;
end if;
-- end of statement processing
end;

```

Figure 9-4: Dynamic SQL Application Fragments

9.2. SQL and Ada Tasks

This section delineates issues arising from the use of SQL within an Ada application using Ada tasking. The issues stem from both practical and theoretical aspects of concurrency control.

The tasks within an Ada multi-tasking program form a set of mutually cooperating sequential programs. The cooperation is mediated by shared variables and rendezvous. The transactions executing concurrently against a shared database form a set of mutually non-interfering sequential programs. The non-interference is mediated by the DBMS's concurrency control protocol, typically locking. The difference between these two views of concurrency is profound. Whereas the purpose of an Ada task control monitor is, in part, to ensure that inter-task communication and cooperation proceed smoothly, the purpose of a DBMS concurrency control monitor is to ensure that inter-transaction communication does not occur at all. The difference in the meaning of correctness of concurrent execution of Ada tasks and DBMS transactions requires that Ada multi-tasking DBMS applications be carefully designed. In particular, the mapping between Ada tasks and DBMS transactions must be carefully considered.

A task is said to be directly associated with a transaction if the task executes a statement of the transaction,³⁷ by way of an abstract procedure call. A task is indirectly associated with a transaction if it causes the execution of such a statement within a task that is directly associated with the transaction. (There may be tasks which are neither directly nor indirectly associated with any transaction.) A mapping between tasks and transactions is a relation which gives the tasks and their associated transactions at some point during the execution of the program. (An application may terminate and restart transactions during its execution. Such sequences of transactions which do not overlap in time present no difficulties. The design and coding difficulties arise in connection with sets of concurrent transactions associated with a single Ada program.) This mapping can be of one of four classes.

1. **One-to-one.** A task is associated, directly or indirectly, with at most one transaction; a transaction is associated with exactly one task.
2. **Many-to-one.** A task is associated with at most one transaction; a transaction is associated with any (positive) number of tasks.
3. **One-to-many.** A task is associated with any number of transactions; each transaction is associated with exactly one task.
4. **Many-to-many.** The mapping between tasks and transactions is unconstrained.

Since a DBMS considers a transaction to be a sequential program, it cannot tolerate concurrent execution of multiple requests on behalf of a single transaction.³⁸ In other words, if either of the relations many-to-one or many-to-many between tasks and transactions is desired, the many tasks associated with any transaction must all use a synchronization or ser-

³⁷The means by which a DBMS identifies the transaction on behalf of which a statement is to be executed is a central issue which will be discussed.

³⁸There are research DBMS prototypes which allow overlapped execution of database operations within the context of a single task. It is highly probable that no commercially available DBMS supports such processing.

vice task to control database operations for that transaction. If an Ada multi-tasking program is to appear to the database as a single transaction at every point in its execution, provision of this synchronization task is all that is required.

The synchronization task can be designed so as to contain the abstract module(s) for all of the tasks associated with the synchronization task's transaction. This may well be a poor design choice. In particular, it may give rise to an inordinate number of task entries. Alternatively, each task within the transaction may contain its own abstract module. The synchronization task provides a semaphore service. Calls to the semaphore task's entries belong in the application, as the abstract module deals only with database interaction and should not be aware of task structure. The semaphore should be acquired before each call to the abstract module's procedures and released upon return. This will ensure that the global SQLCODE variable in SQL_Communications_Pkg, which will be shared by the tasks, is accessed in the critical region delineated by the get and release calls to the semaphore.

If an Ada program is designed to present multiple, concurrent transactions to the DBMS, careful consideration must be given to the semantics of this situation. For simplicity, assume exactly two tasks, T1 and T2, each associated with exactly one transaction, N1 and N2. The DBMS will schedule the operations of N1 and N2 such that they are *serializable*. This is to say that, given the information available to the DBMS, which is exactly the sequence of DBMS operations within N1 and N2, the DBMS will schedule those operations so that their net effect is identical to the effect of executing one of those sequences in its entirety followed by the entirety of the other sequence. In short, serializability provides to each DBMS transaction the illusion that it is running by itself, without competing, concurrent transactions. Now suppose that T1 and T2 share information, through global variables or rendezvous; that the information they share is derived from the database operations they execute; and that the database operations they execute are determined by the information they share. In this case, T1 and T2 cannot be serialized; their net effect is not equivalent to their complete, non-parallel execution in any order. However, that fact is unknown to the DBMS. It may well be that this scenario is not erroneous. That will depend on the semantics of the tasks' interaction. But it must be carefully reviewed.

Cooperating tasks presenting distinct transactions to the DBMS, such as T1 and T2 in the prior paragraph, must be able to deal with each other's abnormal termination. A DBMS may abnormally terminate a well formed, semantically correct transaction in order to resolve a detected deadlock. If, for example, T2 has given information derived from the database to T1, and its associated transaction, N2, is abnormally terminated by the DBMS, the DBMS will not abnormally terminate N1, since it does not know that the communication has taken place. T1 must be able to detect that situation and take whatever action is appropriate.³⁹

The discussion so far has centered on the theoretical issues involved in forming semantically correct multi-tasking, multi-transaction Ada DBMS applications. An example of such a well-formed application is the case of multiple task executions of the same task type, each execution operating on behalf of a distinct user, without inter-task-object communication. The remainder of this section deals with the practical aspects of constructing such well-formed applications.

³⁹This situation is not unique to DBMS applications. Any set of cooperating tasks must be able to deal with each other's abnormal termination.

It must be noted immediately that neither the current ANSI standard [2], nor the follow-on standard [3], allow for the construction of multi-transaction programs. This is because there is no way in the standard to associate a statement execution with a particular transaction among a concurrent set of transactions. This topic will be addressed below.

The ability to construct multi-transaction Ada programs depends in large measure on the target DBMS. There are many things to consider. Every Ada DBMS application will contain in its executable image some code supplied by the DBMS. This code will be called the *DBMS stub*. The function of this stub is to accept the DBMS call from the concrete module and transfer control to the DBMS, which, in a multi-user operating environment, may be executing as a separate process, in a separate address space, or even on a separate machine. It must be the case that either this stub code is reentrant, that is, capable of executing multiple, parallel threads of control, or that each task associated with each transaction has its own, private copy of that code. If neither of these things can be done, multi-transaction programs cannot be written. The same reasoning holds for the concrete module, if distinct tasks, directly associated with distinct transactions, are to share an abstract, and therefore also a concrete, module.

If the reentrancy requirements of the previous paragraph are met by the target DBMS, the final obstacle is the means by which the DBMS identifies the transaction on whose behalf a given statement is to be executed. In the case of a single user DBMS, as might be found on a PC class machine, all statement executions are part of the same transaction, and multi-transaction programs cannot be written. If a multi-user DBMS identifies transactions on the basis of the identity of the program executing the statement, using operating system features to make that identification, multi-transaction programs are again impossible. If the DBMS identifies the transaction by some parameter of the call itself, such as the address of a "communication area," then this parameter can be called a *transaction identifier*. Transaction identifiers do not appear in SQL statements. Dynamic modification of that parameter requires understanding of, and possibly modification to, the code generated by an SQL preprocessor or concrete module compiler, particularly in the case where that concrete module code is to be shared by task objects. This is a tricky and dangerous business, which can result in engineering nightmares.⁴⁰

One way to ensure that task objects do not share abstract or concrete modules is to place these modules within the bodies of the tasks. If the task objects are to logically (but not physically) share an abstract module, the module can be made into a parameterless generic which is instantiated into the task body. If the DBMS identifies transactions via a transaction identifier generated by the SQL processor, this solution may work, at the expense of increased object code size on most compilers. This solution will probably not work to solve reentrancy problems for the DBMS stub code referenced earlier. That code is usually brought into the executable by the system linker, which normally resolves references by name, thereby sharing one copy of the stub among all the tasks.

If multi-transaction programs are not prohibited by any of these considerations, then such programs can be written if a minor modification is made to the standard SAME support packages. In particular, the package `SQL_Communications_Pkg` presents a difficulty as it exports a global variable, `SQLCODE`. This variable can be made local to a task object by

⁴⁰It may be that a DBMS extends SQL to provide a transaction identifier. The author knows of no such DBMS.

the method of the prior paragraph, i.e., by placing this package, along with the abstract and concrete modules and the package SQL_Database_Error_Pkg, into the body of the tasks. If that is otherwise not necessary or desirable, then the package SQL_Communication_Pkg and the calling conventions at the abstract module level (and the concrete level as well, in a non-standard way, see the previous discussion), can be modified as follows: Remove the variable SQLCODE from the specification of SQL_Communications_Pkg and replace it with the following type definition:

```
type Transaction_Id_Type is record
  SQLCODE : SQLCODE_Type;
  <implementation dependent private record type>
end record;
```

(The implementation-dependent portion of the type Transaction_Id_Type is meant to accommodate an implementation defined "communications area." Such an object may also be added to the definition of SQL_Communications_Pkg in the single transaction case.) Each task object directly associated with a transaction must allocate an object of this type in a manner which will allow it to persist across all abstract module procedure calls. The parameter lists of such calls are extended to include that object, which is a transaction identifier. The procedure Process_Database_Error in SQL_Database_Error_Pkg is also amended to include this parameter. Any handler for for the SQL_Database_Error exception must be able to find the appropriate transaction identifier.

References

- [1] *Alsys Ada Sun Workstations Appendix F Version 3.0*
Alsys Inc., Waltham, MA, 1987.
- [2] *Database Language - SQL*
American National Standards Institute, 1986.
X3.135-1986.
- [3] *American National Standard Embedding of SQL Statements into Programming Languages (proposed draft)*
Technical Committee X3H2 - Database, 1988.
X3.168-198x.
- [4] *ISO-ANSI Working Draft Database Language SQL2*
American National Standards Institute, 1987.
X3.135-1986.
- [5] *American National Standard for Information Systems Database Language Embedded SQL (proposed draft)*
Technical Committee X3H2 - Database, 1988.
X3H2-88-320.
- [6] ANSI/X3/Sparc.
Interim Report from the Study Group on Data Base Management Systems.
Bulletin of the ACM SIGMOD 7(2), 1975.
- [7] Chen, P. P-S.
The entity-relationship model; toward a unified view of data.
ACM Transactions on Database Systems 1(1), 1976.
- [8] Clemons, Eric K.
Data Models and the ANSI/SPARC Architecture.
In S. Bing Yao (editor), *Principles of Database Design*, pages 66-114. Prentice Hall, 1985.
- [9] Date, C. J.
An Introduction to the ANSI SQL Standard.
Addison-Wesley Publishing Co., Reading, MA, 1988.
- [10] Felts, Steve.
X3H2 SQL2 Change Proposal: Dynamic SQL Functional Interface.
ANSI X3H2, 1988.
X3H2-88-318 corrected.
- [11] Engle, C; Firth, R.; Graham, M.; Wood, W.
Interfacing Ada and SQL.
Technical Report CMU/SEI-87-TR-48, DTIC: ADA199634, Software Engineering Institute, December, 1987.
- [12] Brykczynski, W.; Friedman, F.; Hilliar, K; Hook, A.
Level 1 Ada/SQL Database Language Interface User's Guide.
Technical Report M-30, Institute for Defense Analyses, September, 1987.
- [13] *Ingres/SQL Reference Manual*
Relational Technology, Inc., 1986.

- [14] Graham, Marc H.
SAME Standard Package Installation Guide
Software Engineering Institute, 1988.
CMU/SEI-89-SR-5.

- [15] *Reference Manual for the Ada Programming Language*
United States Department of Defense, ANSI/MIL-STD-1815A-1983.
American National Standards Institute.

- [16] Shaw, P.
Ada-SQL Interface: Changes in the SQL module language for Ada and deletion of the Ada-SQL embedded syntax.
Technical Report ANSI X3H2-88-182, SQL Ada Module Extensions Design Committee (SAME - DC), May, 1988.

- [17] *VADS UNIX Implementation Reference (Including Ada RM Appendix F)*
Verdix Corporation, 1987.

A SAME Quick Reference List

A.1 Example Domains

Let *Dom* be an abstract domain name for the SQL <type> domains for int, smallint, real, and double_precision.

```
with SQL_<type>_Pkg; use SQL_<type>_Pkg;
...
type Dom_Not_Null is new SQL_<type>_Not_Null;
type Dom_Type is new SQL_<type>;
package Dom_Ops is new SQL_<type>_Ops(Dom_Type, Dom_Not_Null);
```

Let *Dom* be an abstract domain name for the SQL Character domain. In the following example, *n* represents the number of characters in the _Not_Null portion of the domain.

```
with SQL_Char_Pkg; use SQL_Char_Pkg;
...
type DomNN_Base is new SQL_Char_Not_Null;
subtype Dom_Not_Null is DomNN_Base(1..n);
type Dom_Base is new SQL_Char;
subtype Dom_Type is Dom_Base(Dom_Not_Null'Length);
package Dom_Ops is new SQL_Char_Ops(Dom_Type, Dom_Not_Null);
```

Let *Dom* be an abstract domain name for an SQL enumeration domain.

```
with SQL_Enumeration_Pkg;
...
type Dom_Not_Null is (literal, literal, ..., literal);
package Dom_Pkg is new SQL_Enumeration_Pkg (Dom_Not_Null);
type Dom_Type is new Dom_Pkg.SQL_Enumeration;
```

Let *Dom* be an abstract domain name for an SQL Decimal domain. Let the scale of the domain be *s*.

```
with SQL_Decimal_Pkg, Ada_BCD_Pkg;
use SQL_Decimal_Pkg, Ada_BCD_Pkg;
...
Dom_Scale : constant decimal_digits := s;
type DomNN_Base is new SQL_Decimal_Not_Null;
subtype Dom_Not_Null is DomNN_Base(scale => Dom_Scale);
type Dom_Base is new SQL_Char;
subtype Dom_Type is Dom_Base(scale => Dom_Scale);
package Dom_Ops is new SQL_Char_Ops(Dom_Type,
                                     in_scale => Dom_Scale);
```

See Chapter 3 for further details.

A.2 Functions Available to the Application

	Operand Type			Exceptions
	Left	Right	Result	
All Domains				
Null_SQL <type>			_Type	
With_Null		_Not_Null	_Type	
Without_Null		_Type ¹	_Not_Null ²	Null_Value_Error
Is_Null, Not_Null		_Type	Boolean	
Assign ³	_Type	_Type		Constraint_Error
Equals, Not_Equals	_Type	_Type	B_W_U ⁴	
<, >, <=, >=	_Type	_Type	B_W_U	
=, /=, >, <, >=, <=	_Type	_Type	Boolean	
Numeric Domains				
unary +, -, Abs		_Type	_Type	
+, -, /, *	_Type	_Type	_Type	
**	_Type	Integer	_Type	
Int and Smallint Domains				
Mod, Rem	_Type	_Type	_Type	
Image	_Type		SQL_Char	
Image	_Not_Null		SQL_Ch>NN ⁵	
Value	SQL_Char		_Type	
Value	SQL_Ch>NN		_Not_Null	
Decimal Domains				
=, /=, >, <, >=, <=	_Not_Null	_Not_Null	Boolean	
unary +, -, abs		_Not_Null	_Not_Null	
+, -, *, /	_Not_Null	_Not_Null	_Not_Null	Constraint_Error
*, / ⁶	_Not_Null	SQL_Int>NN	_Not_Null	Constraint_Error
*, /	_Type	SQL_Int>NN	_Type	Constraint_Error
*, /	_Type	SQL_Int	_Type	Constraint_Error
*	SQL_Int>NN	_Not_Null	_Not_Null	Constraint_Error
*	SQL_Int>NN	_Type	_Type	Constraint_Error
*	SQL_Int	_Type	_Type	Constraint_Error
Zero, One			_Not_Null	
Zero, One			_Type	
Assign ³	_Not_Null	_Not_Null		Constraint_Error
Shift	_Not_Null	Integer	_Not_Null	Constraint_Error
Shift	_Type	Integer	_Type	Constraint_Error
Width	_Not_Null		Integer	
Width	_Type		Integer	Null_Value_Error
Fore, Aft	_Not_Null		Integer	
Fore, Aft	_Type		Integer	Null_Value_Error
Integral, Scale	_Not_Null		Integer	
Integral, Scale	_Type		Integer	Null_Value_Error
Is_In	_Not_Null		Boolean	
Is_In	_Type		Boolean	

Operand Type

Exceptions

	Left	Right	Result	
Decimal Domains (cont.)				
Machine_Rounds		_Not_Null	Boolean	
Machine_Rounds		_Type	Boolean	
Machine_Overflows		_Not_Null	Boolean	
Machine_Overflows		_Type	Boolean	
To_SQL_Decimal_Not_Null		SQL_IntNN	_Not_Null	
To_SQL_Decimal_Not_Null ⁷		SQL_DblNN	_Not_Null	Constraint_Error
To_SQL_Decimal_Not_Null		SQL_ChrNN	_Not_Null	Constraint_Error
To_SQL_Decimal		SQL_IntNN	_Type	
To_SQL_Decimal		SQL_Int	_Type	
To_SQL_Decimal		SQL_DblNN	_Type	Constraint_Error
To_SQL_Decimal ⁸		SQL_Dbl	_Type	Constraint_Error
To_SQL_Decimal		SQL_ChrNN	_Type	Constraint_Error
To_SQL_Decimal		SQL_Char	_Type	Constraint_Error
To_SQL_Int_Not_Null		_Not_Null	SQL_IntNN	Constraint_Error
To_SQL_Int_Not_Null		_Type	SQL_IntNN	Constraint_Error
				Null_Value_Error
To_SQL_Int		_Type	SQL_Int	Constraint_Error
To_SQL_Double_Precision_Not_Null		_Not_Null	SQL_DblNN	
To_SQL_Double_Precision_Not_Null		_Type	SQL_DblNN	Null_Value_Error
To_SQL_Double_Precision		_Type	SQL_Dbl	
To_SQL_Char_Not_Null		_Not_Null	SQL_ChrNN	
To_SQL_Char_Not_Null		_Type	SQL_ChrNN	Null_Value_Error
To_SQL_Char		_Type	SQL_Char	
To_String		_Not_Null	String	
To_String		_Type	String	Null_Value_Error

Character Domains

Without_Null_Unpadded		_Type	_Not_Null	Null_Value_Error
To_String		_Not_Null	String	
To_String		_Type	String	Null_Value_Error
To_Unpadded_String		_Not_Null	String	
To_Unpadded_String		_Type	String	Null_Value_Error
To_SQL_Char_Not_Null		String	_Not_Null	
To_SQL_Char		String	_Type	
Unpadded_Length		_Type	SQL_U_L ⁹	Null_Value_Error
Substring ¹⁰		_Type	_Type	Constraint_Error
&	_Type	_Type	_Type	

Enumeration Domains

Pred, Succ		_Type	_Type	
Image		_Type	SQL_Char	
Image		_Not_Null	SQL_ChrNN	
Pos		_Type	Integer	Null_Value_Error
Val		Integer	_Type	
Value		SQL_Char	_Type	
Value		SQL_ChrNN	_Not_Null	

	Operand Type			Exceptions
	Left	Right	Result	
Boolean Functions				
not		B_W_U	Boolean	
and, or, xor	B_W_U	B_W_U	Boolean	
To_Boolean		B_W_U	Boolean	Null_Value_Error
Is_True,	B_W_U	B_W_U	Boolean	
Is_False,	B_W_U	B_W_U	Boolean	
Is_Unknown	B_W_U	B_W_U	Boolean	

1. "_Type" represents the type in the abstract domain of which objects that may be null are declared.
2. "_Not_Null" represents the type in the abstract domain of which objects that are not null may be declared.
3. "Assign" is a procedure. The result is returned in object "Left."
4. "B_W_U" is an abbreviation for Boolean_With_Unknown.
5. "SQL_ChrNN" is an abbreviation for SQL_Char_Not_Null.
6. "SQL_IntNN" is an abbreviation for SQL_Int_Not_Null.
7. "SQL_DblNN" is an abbreviation for SQL_Double_Precision_Not_Null.
8. "SQL_Dbl" is an abbreviation for SQL_Double_Precision.
9. "SQL_U_L" is an abbreviation for the SQL_Char_Pkg subtype SQL_Unpadded_Length.
10. Substring has two additional parameters: Start and Length, which are both of the SQL_Char_Pkg subtype SQL_Char_Length.

B Glossary of Terms

Abstract domain. A real world collection of values. Differs from both an Ada type and an SQL type in that it is a real world object, not a programming object. An abstract domain is represented in an Ada program by a pair of type definitions and a generic package instantiation. One of the types, the `_Not_Null` type, can represent any value in the abstract domain except the null value. The other type, the `_Type`, can represent the null value as well. The two types are syntactically connected through the convention of having the same prefix. That is, the abstract domain *Domain* is represented by the two Ada types *Domain_Not_Null* and *Domain_Type*. The two types are semantically connected through the instantiation of an `_Ops` package. See `_Not_Null` type, `_Type` type, `_Ops` package, and Visible Ada type.

Abstract interface. The specification of the abstract module. Contains the declarations of row record types and of abstract procedures. See Abstract module, Abstract procedure, Row record type.

Abstract module. The body of the abstract interface. Contains the bodies of the abstract procedures. See Abstract interface and Abstract procedure.

Abstract procedure. The procedure called by the application program to perform database interaction. The abstract procedure calls the concrete procedure to perform the interaction. The abstract procedure does error checking by examining the `SQLCODE` variable and takes action as necessary. It also does data conversion from concrete to abstract types. See Abstract interface, Abstract module, Concrete procedure, `SQLCODE`, and Standard error processing.

Ada semantics. Refers to the operations predefined in Ada for arithmetic, comparison, etc.

Ada typing model. The ability, in Ada, for the programmer to define new types from existing types. The phrase also refers to Ada's use of name equivalence, rather than structural equivalence, to determine object typing. As two integer types with the same integer range constraint being nonetheless distinct. Ada's typing model also includes so-called "strong" typing.

Application program. The part of the complete application which contains that part of the application logic that is written in Ada. It contains none of the application logic written in SQL, nor any of the bookkeeping logic for executing the SQL. See Concrete module and Abstract module.

Attribute. See Column.

_Base type. Within the definition of a string-based abstract domain, the unconstrained types. The `_Not_Null` and `_Type` types are subtypes of the `_Base` types. See SQL String processing, `_Not_Null` type, and `_Type` type.

Column. A field of a row within a table. Corresponds to Ada's scalar variable in that a field must hold an atomic value and may not contain a composite value. (Character strings are thought of as atomic in this sense.)

Concrete interface. Specification of the concrete module. Contains the declarations of the concrete procedures. See Concrete module and Concrete procedure.

Concrete module. Contains the bodies of the concrete procedures. See Concrete interface and Concrete procedure.

Concrete procedure. A procedure in the concrete module. Concrete procedures perform database interaction. Each concrete procedure corresponds to a single SQL statement.

Concrete types. The types defined in SQL_Standard. These types describe the representation of data in the database.

Comparison rule. A heuristic for determining if two values, variables, or columns have the same type or abstract domain. The rule: *If it makes sense to compare the values, variables or columns, then they have the same type or abstract domain. If it makes no sense to compare them, then they have different types or domains.*

Cursor. Used by SQL to communicate with application languages. A cursor is associated with a Select...From...Where block. A cursor may be opened, fetched, and closed. See an SQL description (e.g., *Database Language - SQL* [2]) for details. A cursor is a quasi-object in that it can be updated and it has state, but it is not available for any programming operations other than SQL statements. The state of a cursor is closed or open; an open cursor records a current position (row) within the associated table. The current row may be deleted or updated.

Database exceptional condition. Any condition which causes SQLCODE to be set to a non-zero value upon return from a concrete procedure. Includes "no record found." Exceptional conditions may be expected or unexpected. See Result parameter and Standard error processing.

Data integrity constraints. Statements made about the contents of the database that are enforced by the database management system.

Data semantics. The meaning of the operations defined on a set of values. See Ada semantics and SQL semantics.

Derived type. A type whose operations and values are replicas of those of an existing type. The existing type is called the parent type of the derived type. *LRM* glossary [15].

Domain package. An Ada package specification containing only declarations of abstract domains. No abstract domain declaration may appear in more than one domain package, and no abstract domain declaration may appear outside of a domain package. See Abstract domain.

Dynamic SQL. A form of SQL in which the statement to be executed is created by the application at run time. Dynamic SQL is used when a database interaction takes parameters which are not constants. These can be search conditions, table names, etc.

Full SQL treatment of nulls. The discipline of handling null values in Ada programs that use SQL semantics for arithmetic and comparison operators. This discipline treats variables of `_Type` type as regular variables, using the versions of arithmetic and comparison operators exported by the SAME standard packages.

Indicator parameters. Special integer-typed parameters used at the concrete interface to record information about other parameters. A negative indicator parameter value indicates a null value in the associated parameter. Indicator parameters do not appear at the abstract interface.

Minimalist treatment of nulls. The discipline for handling null values in an Ada program that uses only test (`Is_Null`, `Not_Null`) and conversion (`With_Null`, `Without_Null`) functions. Treats variables of `_Type` type as value repositories only. See `_Type` type, Full SQL treatment of nulls.

Modular approach. Any technique for constructing DBMS application software which physically separates the database interaction statements and the programming language statements.

Module. A related set of procedures which perform database interaction. See Abstract Module, and Concrete Module.

Module Language. The language in which SQL modules are written. Part of ANSI standard SQL. The module language describes procedures, the bodies of which are single SQL statements.

`_Not_Null` type. One of the two types making up an abstract domain definition; so-called because the set of objects of this type does not include the null value. Usually, the `_Not_Null` type is a visible Ada type. See Abstract domain and Visible Ada type.

Null value. SQL's means of recording missing information. A null value in a column indicates that nothing is known about the value which should occupy the column.

`_Ops` generic package. Each of the SAME standard packages contains a generic subpackage which generates, by package instantiation, those functions or procedures that cannot be produced by subprogram derivation. The subpackage name is formed by replacing the `_Pkg` suffix in the containing package name with `_Ops`. In use, the `_Ops` package takes two types as formal parameters, the `_Type` and `_Not_Null` types, which together make up the abstract domain definition.

Platform, or platform specific. The platform on which a piece of software runs is the combination of the hardware, operating system, DBMS and Ada compiler. Pieces of the SAME which are platform specific are the database layer, containing the packages `SQL_System` and `SQL_Standard`, to describe concrete DBMS types in Ada, `SQL_Communications_Pkg`, for retrieving and storing status information from the DBMS, and `SQL_Database_Error_Pkg`, for reporting errors.

Result parameter. An optional parameter, of an enumeration type, frequently Boolean, to every abstract procedure declaration. If present, the result parameter is used by an abstract procedure to signal the occurrence of an expected exceptional condition. See DBMS exceptional condition.

Row. An element of a table. Also called a *tuple*. Analogous to a record object. See Column and Table.

Row record. The object returned from an abstract procedure which retrieves data from the database. Also, the object given to an abstract procedure which stores data in the database. A row record contains a field for each element in the target list of the SQL statement executed by the abstract procedure.

Row record type. The Ada type definition of the row record. Declared in the abstract interface.

SAME standard packages. The packages which support the SAME method; particularly, those packages which support SQL data semantics. Those packages are SQL_Int_Pkg, SQL_Smallint_Pkg, SQL_Real_Pkg, SQL_Double_Precision_Pkg, and SQL_Char_Pkg, which provide support for the standard SQL data types. Other standard SAME packages are SQL_System, SQL_Standard, SQL_Exceptions, SQL_Boolean_Pkg, SQL_Communications_Pkg, and SQL_Database_Error_Pkg. See Platform, SQL semantics, Standard error processing, and User-defined semantics.

SQLCODE. The name of the parameter to a concrete procedure which holds the status code at procedure termination. Also references the values of the parameter.

SQL module. A concrete module written in the module language.

SQL procedure. A procedure defined within the concrete module whose semantics are given by an SQL statement. See Concrete module, Module language, and SQL module.

SQL semantics. The operations of arithmetic and comparison extended to cover the null value. Refers also to SQL string processing, in which strings are automatically padded or truncated during comparisons and assignments. See Three-valued logic and Three-valued arithmetic.

SQL String Processing. SQL treats character strings as fixed length objects in some circumstances and variable length objects in others. For example, all string objects within a given database column have the same length which is given by the column definition. However, when transporting data between and application and the database, an SQL DBMS will truncate or blank pad a string value, as appropriate to the length of the programming language variable. When comparing strings of different lengths, SQL pads the shorter string with blanks before the compare. The SAME standard support package SQL_Char_Pkg offers an Ada implementation of these semantics. See _Base type, _Type type, _Not_Null type.

Standard error processing. The process initiated after an unexpected exceptional condition arises: Process_Database_Error in package SQL_Database_Error_Pkg is called and an exception, SQL_Database_Error, defined in SQL_Communications_Pkg, is raised.

Status parameter. See Result parameter.

Three-valued arithmetic. The arithmetic operations within SQL which are defined to cover the null value. Three-valued arithmetic operations act just like their normal counterparts on non-null values; they return the null value if any of their operands are null.

Three-valued logic. The extension of comparison and Boolean operations within SQL to cover null values. SQL comparison operations return the truth value UNKNOWN if either of their operands are null. SQL defines Boolean operations (and, or, not) on the three-valued set of Boolean operands [FALSE, UNKNOWN, TRUE].

Transaction. A logic unit of database work. Database transaction control provides *transaction atomicity*; i. e., (1) either all of the database modifications performed by any transaction occur or none of them do, and (2) the effect of every successful transaction is the same, whether or not other transactions are executing concurrently.

_Type type. One of the two types making up an abstract domain definition. The set of objects of this type includes the null value. Usually, the _Type type is a private record type. See Abstract domain.

User-defined semantics. The semantics of operators supplied by support packages written by users. These packages allow users to the SAME to fit local needs.

Visible Ada type. Opposite of a private type. See _Not_Null type.

C SAME Standard Package Listings

C.1 Introduction

This appendix contains the source code of the SAME standard packages. This code will be available in machine-readable form from the SEI for a limited time. Please read the copyright notice in the next section. A copy of this notice appears in each file of the machine-readable distribution.

Every procedure and function declaration in these packages is followed by a **pragma IN-LINE** which has been "commented out." The explanation for this is as follows. Almost all of the procedures and functions in these packages are extremely small. Many consist of a single **if** or **return** statement. Therefore they are excellent candidates for procedure inlining which will decrease their runtime cost by the overhead of a procedure call. Experience in using this code with various compilers has shown that this degree of inlining tends to uncover compiler errors and produce inexplicable timings. The safest approach, that of not using inlining at all, has been chosen for the code as distributed. The installer is urged to experiment with the inlining of this code. Some experiments have shown a tenfold speedup due to inlining (whereas other experiments, on other compilers and machine architectures, showed marginal slowdown due to inlining). Recall that inlining will usually make the resulting object module larger.

C.2 Copyright Notice

```
-- ++++++
--
-- The following copyright must be included in this software and
-- all software utilizing this software.
--
-----
-- Copyright (C) 1988 by the Carnegie Mellon University, Pittsburgh, PA.
-- The Software Engineering Institute (SEI) is a federally funded
-- research and development center established and operated by Carnegie
-- Mellon University (CMU). Sponsored by the U.S. Department of Defense
-- under contract F19628-85-C-0003, the SEI is supported by the
-- services and defense agencies, with the U.S. Air Force as the
-- executive contracting agent.
--
-- Permission to use, copy, modify, or distribute this software and its
-- documentation for any purpose and without fee
-- is hereby granted, provided
-- that the above copyright notice appear in all copies and that both
-- that copyright notice and this permission notice appear in supporting
-- documentation. Further, the names Software Engineering Institute or
-- Carnegie Mellon University may not be used in advertising or publicity
-- pertaining to distribution of the software without specific, written
-- prior permission. CMU makes no claims or representations
-- about the suitability of
-- this software for any purpose. This software is provided "as is"
-- and no warranty, express or implied, is made by the SEI or CMU,
-- as to the accuracy
-- and functioning of the program and related program material, nor
-- shall the fact of distribution constitute any such warranty. No
-- responsibility is assumed by the SEI or CMU in connection herewith.
-----
--
```

C.3 SQL_System Specification

```
-- SQL_System is a "platform-specific" package
-- within the SAME

package SQL_System is

    -- MAXCHRLen is the length of the longest character string
    -- which the DBMS will store.
    -- It serves as the upper bound on SQL_Char_Pkg
    -- subtypes SQL_Char_Length and SQL_Unpadded_Length.
    -- SQL_Char_Length is a subtype of Natural with a lower bound
    -- of 1.
    -- SQL_Unpadded_Length is a subtype of Natural with a lower
    -- bound of 0.

    MAXCHRLen : constant integer := str_length; -- replace

    -- MAXERRLEN is the maximum length of the error message
    -- string returned from DBMS specific error message routine

    MAXERRLEN : constant integer := msg_length; -- replace

end SQL_System;
```

C.4 SQL_Standard Specification

```
package Sql_Standard is
  package Character_Set renames cst;
  subtype Character_Type is Character_Set.cst;
  type Char is array (positive range <>)
    of Character_Type;
  type Smallint is range bs..ts;
  type Int is range bi..ti;
  type Real is digits dr;
  type Double_Precision is digits dd;
  -- type Decimal is to be determined;
  type Sqlcode_Type is range bsc..tsc;
  subtype Sql_Error is Sqlcode_Type
    range Sqlcode_Type'FIRST .. -1;
  subtype Not_Found is Sqlcode_Type
    range 100..100;
  subtype Indicator_Type is t;

  -- csp is an implementor-defined package and cst is an
  -- implementor-defined character type. bs, ts, bi, ti, dr, dd, bsc,
  -- and tsc are implementor defined integral values. t is int or
  -- smallint corresponding to an implementor-defined <exact
  -- numeric type> of indicator parameters.

end sql_standard;
```

C.5 SQL_Communications_Pkg Specification

```
with SQL_Char_Pkg; use SQL_Char_Pkg;
with SQL_Standard; use SQL_Standard;
package SQL_Communications_Pkg is

  -- This is an example of the package, providing minimal functionality.
  -- This package may be tailored to the needs of a given platform.

  SQL_Database_Error : exception;

  SQLCODE : SQLCODE_TYPE;

  -- Parameterless function returning an error message of type
  -- SQL_Char_Not_Null.
  -- The error message is the descriptive string associated with
  -- the most recent database error. It is produced by a
  -- DBMS specific function.

  function SQL_Database_Error_Message return SQL_Char_Not_Null;

end SQL_Communications_Pkg;
```

C.6 SQL_Communications_Pkg Body

```
-- SQL_Communications_Pkg is a "platform-specific" package
-- within the SAME
-- this particular version of the package was developed for
-- a platform consisting of the Verdix (Version 5.41) Ada compiler
-- and INGRES (Version 5.0) running on a Vax Station
```

```

with system; use system;
with SQL_System; use SQL_System;
with ingres_c_support; use ingres_c_support;
-- ingres_c_support contains functions Add_Null and Strip_Null
-- which are used to convert between 'c' format strings and
-- Ada format strings. It is not included in the SAME standard packages.
package body SQL_Communications_Pkg is

```

```

function SQL_Database_Error_Message return SQL_Char_Not_Null is

```

```

    Message_Buffer : SQL_Char_Not_Null (1..MAXERRLEN);

```

```

    Len : integer := MAXERRLEN;

```

```

    procedure geterrmsg (Message : in Address;
                        Length : in Address);

```

```

    pragma interface(C, geterrmsg, "_sqlerrmsg");

```

```

begin

```

```

    geterrmsg (Message_Buffer'Address, Len'Address);

```

```

    -- the assumption here is that no error will occur when
    -- retrieving the error message from the database

```

```

    return strip_null(Message_Buffer);

```

```

end SQL_Database_Error_Message;

```

```

end SQL_Communications_Pkg;

```

C.7 SQL_Exceptions Specification

```

package SQL_exceptions
is

```

```

    Null_Value_Error : exception;

```

```

end SQL_exceptions;

```

C.8 SQL_Boolean_Pkg Specification

```

package SQL_Boolean_Pkg
is

```

```

    type Boolean_with_Unknown is (FALSE, UNKNOWN, TRUE);

```

```

    ---- Three valued Logic operations ----

```

```

    --- three-val X three-val => three-val ---

```

```

    function "not" (Left : Boolean_with_Unknown)
                    return Boolean_with_Unknown;

```

```

    -- pragma INLINE ("not");

```

```

    function "and" (Left, Right : Boolean_with_Unknown)
                    return Boolean_with_Unknown;

```

```

    -- pragma INLINE ("and");

```

```

    function "or" (Left, Right : Boolean_with_Unknown)
                    return Boolean_with_Unknown;

```



```

-- pragma INLINE ("or");
function "xor" (Left, Right : Boolean_with_Unknown)
    return Boolean_with_Unknown;
-- pragma INLINE ("xor");

--- three-val => bool or exception ---
function To_Boolean (Left : Boolean_with_Unknown) return Boolean;
-- pragma INLINE (To_Boolean);

--- three-val => bool ---
function Is_True (Left : Boolean_with_Unknown) return Boolean;
-- pragma INLINE (Is_True);
function Is_False (Left : Boolean_with_Unknown) return Boolean;
-- pragma INLINE (Is_False);
function Is_Unknown (Left : Boolean_with_Unknown) return Boolean;
-- pragma INLINE (Is_Unknown);

end SQL_Boolean_Pkg;

```

C.9 SQL_Boolean_Pkg Body

```

With SQL_Exceptions;
package body SQL_Boolean_Pkg is

    Null_Value_Error : exception renames SQL_Exceptions.Null_Value_Error;

    function "not" (Left : Boolean_with_Unknown)
        return Boolean_with_Unknown is
    begin
        case Left is
            when true => return false;
            when false => return true;
            when unknown => return unknown;
        end case;
    end;

    function "and" (Left, Right : Boolean_with_Unknown)
        return Boolean_with_Unknown is
    begin
        if (Left = False) or else (Right = False) then
            return False;
        elsif (Left = Unknown) or else (Right = Unknown) then
            return Unknown;
        else
            return True;
        end if;
    end;

    function "or" (Left, Right : Boolean_with_Unknown)
        return Boolean_with_Unknown is
    begin
        if (Left = True) or else (Right = True) then
            return True;
        elsif (Left = Unknown) or else (Right = Unknown) then
            return Unknown;
        else
            return False;
        end if;
    end;

```

```

function "xor" (Left, Right : Boolean_with_Unknown)
    return Boolean_with_Unknown is
begin
    return (Left and not Right) or (not Left and Right);
end;

--- three-val => bool or exception ---
function To_Boolean (Left : Boolean_with_Unknown) return Boolean is
begin
    if Left = Unknown then raise null_value_error;
    else return (Left = True);
    end if;
end;

--- three-val => bool ---
function Is_True (Left : Boolean_with_Unknown) return Boolean is
begin
    return (Left = True);
end;
function Is_False (Left : Boolean_with_Unknown) return Boolean is
begin
    return (Left = False);
end;
function Is_Unknown (Left : Boolean_with_Unknown) return Boolean is
begin
    return (Left = Unknown);
end;

end SQL_Boolean_Pkg;

```

C.10 SQL_Int_Pkg Specification

```

with SQL_Standard;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Char_Pkg; use SQL_Char_Pkg;
package SQL_Int_Pkg
    is

    type SQL_Int_not_null is new SQL_Standard.Int;

    ---- Possibly Null Integer ----
    type SQL_Int is limited private;

    function Null_SQL_Int return SQL_Int;
    -- pragma INLINE (Null_SQL_Int);

    -- this pair of functions convert between the
    -- null-bearing and non-null-bearing types.
    function Without_Null_Base(Value : SQL_Int)
        return SQL_Int_Not_Null;
    -- pragma INLINE (Without_Null_Base);
    -- With_Null_Base raises Null_Value_Error if the input
    -- value is null
    function With_Null_Base(Value : SQL_Int_Not_Null)
        return SQL_Int;
    -- pragma INLINE (With_Null_Base);

    -- this procedure implements range checking
    -- note: it is not meant to be used directly
    -- by application programmers
    -- see the generic package SQL_Int_Ops

```

```

-- raises constraint_error if not
-- (First <= Right <= Last)
procedure Assign_with_check (
    Left : in out SQL_Int; Right : SQL_Int;
    First, Last : SQL_Int_Not_Null);
-- pragma INLINE (Assign_with_check);

-- the following functions implement three valued
-- arithmetic
-- if either input to any of these functions is null
-- the function returns the null value; otherwise
-- they perform the indicated operation
-- these functions raise no exceptions
function "+"(Right : SQL_Int) return SQL_Int;
-- pragma INLINE ("+");
function "-"(Right : SQL_Int) return SQL_Int;
-- pragma INLINE ("-");
function "abs"(Right : SQL_Int) return SQL_Int;
-- pragma INLINE ("abs");
function "+"(Left, Right : SQL_Int) return SQL_Int;
-- pragma INLINE ("+");
function "*" (Left, Right : SQL_Int) return SQL_Int;
-- pragma INLINE ("*");
function "-"(Left, Right : SQL_Int) return SQL_Int;
-- pragma INLINE ("-");
function "/"(Left, Right : SQL_Int) return SQL_Int;
-- pragma INLINE ("/");
function "mod" (Left, Right : SQL_Int) return SQL_Int;
-- pragma INLINE ("mod");
function "rem" (Left, Right : SQL_Int) return SQL_Int;
-- pragma INLINE ("rem");
function "***" (Left : SQL_Int; Right: Integer) return SQL_Int;
-- pragma INLINE ("***");

-- simulation of 'IMAGE and 'VALUE that
-- return/take SQL_Char[_Not_Null] instead of string
function IMAGE (Left : SQL_Int_Not_Null) return SQL_Char_Not_Null;
function IMAGE (Left : SQL_Int) return SQL_Char;
function VALUE (Left : SQL_Char_Not_Null) return SQL_Int_Not_Null;
function VALUE (Left : SQL_Char) return SQL_Int;

-- Logical Operations --
-- type X type => Boolean_with_unknown --
-- these functions implement three valued logic
-- if either input is the null value, the functions
-- return the truth value UNKNOWN; otherwise they
-- perform the indicated comparison.
-- these functions raise no exceptions
function Equals (Left, Right : SQL_Int) return Boolean_with_Unknown;
-- pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Int)
    return Boolean_with_Unknown;
-- pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Int) return Boolean_with_Unknown;
-- pragma INLINE ("<");
function ">" (Left, Right : SQL_Int) return Boolean_with_Unknown;
-- pragma INLINE (">");
function "<=" (Left, Right : SQL_Int) return Boolean_with_Unknown;
-- pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Int) return Boolean_with_Unknown;
-- pragma INLINE (">=");

-- type => boolean --

```

```

function Is_Null(Value : SQL_Int) return Boolean;
-- pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Int) return Boolean;
-- pragma INLINE (Not_Null);

-- These functions of class type => boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.
function "=" (Left, Right : SQL_Int) return Boolean;
-- pragma INLINE ("=");
function "<" (Left, Right : SQL_Int) return Boolean;
-- pragma INLINE ("<");
function ">" (Left, Right : SQL_Int) return Boolean;
-- pragma INLINE (">");
function "<=" (Left, Right : SQL_Int) return Boolean;
-- pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Int) return Boolean;
-- pragma INLINE (">=");

-- this generic is instantiated once for every abstract
-- domain based on the SQL type Int.
-- the three subprogram formal parameters are meant to
-- default to the programs declared above.
-- that is, the package should be instantiated in the
-- scope of a use clause for SQL_Int_Pkg.
-- the two actual types together form the abstract
-- domain.
-- the purpose of the generic is to create functions
-- which convert between the two actual types and a
-- procedure which implements a range constrained
-- assignment for the null-bearing type.
-- the bodies of these subprograms are calls to
-- subprograms declared above and passed as defaults to
-- the generic.
generic
type With_Null_type is limited private;
type Without_null_type is range <>;
with function With_Null_Base(Value : SQL_Int+ Not_Null)
return With_Null_Type is <>;
with function Without_Null_Base(Value : With_Null_Type)
return SQL_Int_Not_Null is <>;
with procedure Assign_with_check (
Left : in out With_Null_Type; Right : With_Null_Type;
First, Last : SQL_Int_Not_Null) is <>;
package SQL_Int_Ops is
function With_Null (Value : Without_Null_type)
return With_Null_type;
-- pragma INLINE (With_Null);
function Without_Null (Value : With_Null_Type)
return Without_Null_type;
-- pragma INLINE (Without_Null);
procedure assign (Left : in out With_null_Type;
Right : in With_null_type);
-- pragma INLINE (assign);
end SQL_Int_Ops;

private

type SQL_Int is record
Is_Null: Boolean := true;
Value: SQL_Int_Not_Null;

```

```
end record;
end SQL_Int_Pkg;
```

C.11 SQL_Int_Pkg Body

```
with SQL_exceptions;
package body SQL_Int_pkg is

    Null_Value_Error : exception renames SQL_exceptions.null_value_error;

    function Without_Null_Base(Value : SQL_Int)
    return SQL_Int_Not_Null is
    begin
        if Value.Is_Null then
            raise Null_Value_Error;
        else
            return Value.Value;
        end if;
    end Without_Null_Base;

    function With_Null_Base(Value : SQL_Int_Not_Null)
    return SQL_Int is
    begin
        return(False, Value);
    end With_Null_Base;

    procedure Assign_with_check (
        Left : in out SQL_Int; Right : SQL_Int;
        First, Last : SQL_Int_Not_Null) is
    begin
        if Right.Is_Null then Left.is_Null := True;
        elsif
            (Right.Value < First or else
             Right.Value > Last) then
            raise Constraint_Error;
        else
            Left := Right;
        end if;
    end Assign_With_Check;

    function Null_SQL_Int return SQL_Int is
        Null_Holder : SQL_Int;
    begin
        return (Null_Holder); -- relies on default expression for Is_Null
    end Null_SQL_Int;

    function "+"(Right : SQL_Int) return SQL_Int is
    begin
        return Right;
    end;

    function "--"(Right : SQL_Int) return SQL_Int is
    begin
        return (Right.Is_Null, -(Right.Value));
    end;

    function "abs"(Right : SQL_Int) return SQL_Int is
    begin
        return (Right.Is_Null, abs(Right.Value));
    end;
```

```

end;

function "+"(Left, Right : SQL_Int) return SQL_Int is
begin
  if Left.Is_Null or Right.Is_Null then
    return Null_SQL_Int;
  else
    return (False, (Left.Value + Right.Value));
  end if;
end;

function "*" (Left, Right : SQL_Int) return SQL_Int is
begin
  if Left.Is_Null or Right.Is_Null then
    return Null_SQL_Int;
  else
    return (False, (Left.Value * Right.Value));
  end if;
end;

function "-"(Left, Right : SQL_Int) return SQL_Int is
begin
  if Left.Is_Null or Right.Is_Null then
    return Null_SQL_Int;
  else
    return (False, (Left.Value - Right.Value));
  end if;
end;

function "/"(Left, Right : SQL_Int) return SQL_Int is
begin
  if Left.Is_Null or Right.Is_Null then
    return Null_SQL_Int;
  else
    return (False, (Left.Value / Right.Value));
  end if;
end;

function "mod" (Left, Right : SQL_Int) return SQL_Int is
begin
  if Left.Is_Null or Right.Is_Null then
    return Null_SQL_Int;
  else
    return (False, (Left.Value mod Right.Value));
  end if;
end;

function "rem" (Left, Right : SQL_Int) return SQL_Int is
begin
  if Left.Is_Null or Right.Is_Null then
    return Null_SQL_Int;
  else
    return (False, (Left.Value rem Right.Value));
  end if;
end;

function "**" (Left : SQL_Int; Right: Integer) return SQL_Int is
begin
  if Left.Is_Null then
    return Null_SQL_Int;
  else
    return (False, (Left.Value ** Right));
  end if;

```

```

end;

function IMAGE (Left : SQL_Int_Not_Null) return SQL_Char_Not_Null is
begin
    return to_SQL_Char_Not_Null(SQL_Int_Not_Null'IMAGE(Left));
end IMAGE;

function IMAGE (Left : SQL_Int) return SQL_Char is
begin
    if not Left.Is_Null then
        return to_SQL_Char(SQL_Int_Not_Null'IMAGE(Left.Value));
    else
        return Null_SQL_Char;
    end if;
end IMAGE;

function VALUE (Left : SQL_Char_Not_Null) return SQL_Int_Not_Null is
begin
    return SQL_Int_Not_Null'VALUE(to_String(Left));
end VALUE;

function VALUE (Left : SQL_Char) return SQL_Int is
begin
    if Not_Null(Left) then
        return With_Null_Base(SQL_Int_Not_Null'Value(to_String(Left)));
    else
        return Null_SQL_Int;
    end if;
end VALUE;

-- Logical Operations --
-- type X type => Boolean_with_unknown --
function Equals (Left, Right : SQL_Int) return Boolean_with_Unknown is
begin
    if Left.Is_Null or Right.Is_Null then
        return Unknown;
    else
        if (Left.Value = Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end;

function Not_Equals (Left, Right : SQL_Int)
    return Boolean_with_Unknown is
begin
    if Left.Is_Null or Right.Is_Null then
        return Unknown;
    else
        if (Left.Value = Right.Value) then
            return False;
        else
            return True;
        end if;
    end if;
end;

function "<" (Left, Right : SQL_Int) return Boolean_with_Unknown is
begin
    if Left.Is_Null or Right.Is_Null then
        return Unknown;

```

```

        else
            if (Left.Value < Right.Value) then
                return True;
            else
                return False;
            end if;
        end if;
    end;

function ">" (Left, Right : SQL_Int) return Boolean_with_Unknown is
begin
    if Left.Is_Null or Right.Is_Null then
        return Unknown;
    else
        if (Left.Value > Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end;

function "<=" (Left, Right : SQL_Int) return Boolean_with_Unknown is
begin
    if Left.Is_Null or Right.Is_Null then
        return Unknown;
    else
        if (Left.Value <= Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end;

function ">=" (Left, Right : SQL_Int) return Boolean_with_Unknown is
begin
    if Left.Is_Null or Right.Is_Null then
        return Unknown;
    else
        if (Left.Value >= Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end ">=";

-- type => boolean --
function Is_Null(Value : SQL_Int) return Boolean is
begin
    return Value.Is_Null;
end Is_Null;

function Not_Null(Value : SQL_Int) return Boolean is
begin
    return not Value.Is_Null;
end Not_Null;

function "=" (Left, Right : SQL_Int) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then

```



```

        return FALSE;
    else
        return Left.Value = Right.Value;
    end if;
end "=";
function "<" (Left, Right : SQL_Int) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value < Right.Value;
    end if;
end "<";
function ">" (Left, Right : SQL_Int) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value > Right.Value;
    end if;
end ">";
function "<=" (Left, Right : SQL_Int) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value <= Right.Value;
    end if;
end "<=";
function ">=" (Left, Right : SQL_Int) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value >= Right.Value;
    end if;
end ">=";

package body SQL_Int_Ops is
    function With_Null (Value : Without_Null_Type)
        return With_Null_Type is
    begin
        return (With_Null_Base(SQL_Int_Not_Null(Value)));
    end With_Null;

    function Without_Null (Value : With_Null_Type)
        return Without_Null_Type is
    begin
        return (Without_Null_Type(
            SQL_Int_Not_Null(Without_Null_Base(Value))));
    end Without_Null;

    procedure assign (Left : in out With_Null_Type;
        Right : in With_Null_Type) is
    begin
        Assign_With_Check(Left, Right,
            SQL_Int_Not_Null(Without_Null_Type'FIRST),
            SQL_Int_Not_Null(Without_Null_Type'LAST));
    end assign;

end SQL_Int_Ops;

end SQL_Int_Pkg;

```

C.12 SQL_Smallint_Pkg Specification

```
with SQL_Standard;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Char_Pkg; use SQL_Char_Pkg;
package SQL_Smallint_Pkg
  is
    type SQL_Smallint_not_null is new SQL_Standard.Smallint;

    ---- Possibly Null Integer ----
    type SQL_Smallint is limited private;

    function Null_SQL_Smallint return SQL_Smallint;
    -- pragma INLINE (Null_SQL_Smallint);

    -- this pair of functions converts between the
    -- null-bearing and non-null-bearing types.
    function Without_Null_Base(Value : SQL_Smallint)
      return SQL_Smallint_Not_Null;
    -- pragma INLINE (Without_Null_Base);
    -- With_Null_Base raises Null_Value_Error if the input
    -- value is null
    function With_Null_Base(Value : SQL_Smallint_Not_Null)
      return SQL_Smallint;
    -- pragma INLINE (With_Null_Base);

    -- this procedure implements range checking
    -- note: it is not meant to be used directly
    -- by application programmers
    -- see the generic package SQL_Smallint_Op
    -- raises constraint_error if not
    -- (First <= Right <= Last)
    procedure Assign_with_check (
      Left : in out SQL_Smallint; Right : SQL_Smallint;
      First, Last : SQL_Smallint_Not_Null);
    -- pragma INLINE (Assign_with_check);

    -- the following functions implement three valued
    -- arithmetic
    -- if either input to any of these functions is null
    -- the function returns the null value; otherwise
    -- they perform the indicated operation
    -- these functions raise no exceptions
    function "+"(Right : SQL_Smallint) return SQL_Smallint;
    -- pragma INLINE ("+");
    function "-"(Right : SQL_Smallint) return SQL_Smallint;
    -- pragma INLINE ("-");
    function "abs"(Right : SQL_Smallint) return SQL_Smallint;
    -- pragma INLINE ("abs");
    function "+"(Left, Right : SQL_Smallint) return SQL_Smallint;
    -- pragma INLINE ("+");
    function "*" (Left, Right : SQL_Smallint) return SQL_Smallint;
    -- pragma INLINE ("*");
    function "-"(Left, Right : SQL_Smallint) return SQL_Smallint;
    -- pragma INLINE ("-");
    function "/"(Left, Right : SQL_Smallint) return SQL_Smallint;
    -- pragma INLINE ("/");
    function "mod" (Left, Right : SQL_Smallint) return SQL_Smallint;
    -- pragma INLINE ("mod");
    function "rem" (Left, Right : SQL_Smallint) return SQL_Smallint;
    -- pragma INLINE ("rem");
    function "***" (Left : SQL_Smallint; Right: Integer) return SQL_Smallint;
```

```

-- pragma INLINE ("***");

-- simulation of 'IMAGE and 'VALUE that
-- return/take SQL_Char[Not_Null] instead of string
function IMAGE (Left : SQL_Smallint_Not_Null) return SQL_Char_Not_Null;
function IMAGE (Left : SQL_Smallint) return SQL_Char;
function VALUE (Left : SQL_Char_Not_Null) return SQL_Smallint_Not_Null;
function VALUE (Left : SQL_Char) return SQL_Smallint;

-- Logical Operations --
-- type X type => Boolean_with_unknown --
-- these functions implement three valued logic
-- if either input is the null value, the functions
-- return the truth value UNKNOWN; otherwise they
-- perform the indicated comparison.
-- these functions raise no exceptions
function Equals (Left, Right : SQL_Smallint) return Boolean_with_Unknown;
-- pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Smallint)
    return Boolean_with_Unknown;
-- pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Smallint) return Boolean_with_Unknown;
-- pragma INLINE ("<");
function ">" (Left, Right : SQL_Smallint) return Boolean_with_Unknown;
-- pragma INLINE (">");
function "<=" (Left, Right : SQL_Smallint) return Boolean_with_Unknown;
-- pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Smallint) return Boolean_with_Unknown;
-- pragma INLINE (">=");

-- type => boolean --
function Is_Null (Value : SQL_Smallint) return Boolean;
-- pragma INLINE (Is_Null);
function Not_Null (Value : SQL_Smallint) return Boolean;
-- pragma INLINE (Not_Null);

-- These functions of class type => boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.
function "=" (Left, Right : SQL_Smallint) return Boolean;
-- pragma INLINE ("=");
function "<" (Left, Right : SQL_Smallint) return Boolean;
-- pragma INLINE ("<");
function ">" (Left, Right : SQL_Smallint) return Boolean;
-- pragma INLINE (">");
function "<=" (Left, Right : SQL_Smallint) return Boolean;
-- pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Smallint) return Boolean;
-- pragma INLINE (">=");

-- this generic is instantiated once for every abstract
-- domain based on the SQL type Smallint.
-- the three subprogram formal parameters are meant to
-- default to the programs declared above.
-- that is, the package should be instantiated in the
-- scope of a use clause for SQL_Smallint_Pkg.
-- the two actual types together form the abstract
-- domain.
-- the purpose of the generic is to create functions
-- which convert between the two actual types and a
-- procedure which implements a range constrained

```

```

-- assignment for the null-bearing type.
-- the bodies of these subprograms are calls to
-- subprograms declared above and passed as defaults to
-- the generic.
generic
type With_Null_type is limited private;
type Without_null_type is range <>;
with function With_Null_Base(Value : SQL_Smallint_Not_Null)
return With_Null_Type is <>;
with function Without_Null_Base(Value : With_Null_Type)
return SQL_Smallint_Not_Null is <>;
with procedure Assign_with_check (
Left : in out With_Null_Type; Right : With_Null_Type;
First, Last : SQL_Smallint_Not_Null) is <>;
package SQL_Smallint_OPs is
function With_Null (Value : Without_Null_type)
return With_Null_type;
-- pragma INLINE (With_Null);
function Without_Null (Value : With_Null_Type)
return Without_Null_type;
-- pragma INLINE (Without_Null);
procedure assign (Left : in out With_null_Type;
Right : in With_null_type);
-- pragma INLINE (assign);
end SQL_Smallint_ops;

private

type SQL_Smallint is record
Is_Null: Boolean := true;
Value: SQL_Smallint_Not_Null;
end record;

end SQL_Smallint_Pkg;

```

C.13 SQL_Smallint_Pkg Body

```

with SQL_exceptions;
package body SQL_Smallint_pkg is

Null_Value_Error : exception renames SQL_exceptions.null_value_error;

function Without_Null_Base(Value : SQL_Smallint)
return SQL_Smallint_Not_Null is
begin
if Value.Is_Null then
raise Null_Value_Error;
else
return Value.Value;
end if;
end Without_Null_Base;

function With_Null_Base(Value : SQL_Smallint_Not_Null)
return SQL_Smallint is
begin
return(False, Value);
end With_Null_Base;

procedure Assign_with_check (
Left : in out SQL_Smallint; Right : SQL_Smallint;

```

```

    First, Last : SQL_Smallint_Not_Null) is
begin
    if Right.Is_Null then Left.Is_Null := True;
    elsif
        (Right.Value < First or else
         Right.Value > Last) then
            raise Constraint_Error;
        else
            Left := Right;
        end if;
end Assign_With_Check;

function Null_SQL_Smallint return SQL_Smallint is
    Null_Holder : SQL_Smallint;
begin
    return (Null_Holder); -- relies on default expression for Is_Null
end Null_SQL_Smallint;

function "+"(Right : SQL_Smallint) return SQL_Smallint is
begin
    return Right;
end;

function "-"(Right : SQL_Smallint) return SQL_Smallint is
begin
    return (Right.Is_Null, -(Right.Value));
end;

function "abs"(Right : SQL_Smallint) return SQL_Smallint is
begin
    return (Right.Is_Null, abs(Right.Value));
end;

function "+"(Left, Right : SQL_Smallint) return SQL_Smallint is
begin
    if Left.Is_Null or Right.Is_Null then
        return Null_SQL_Smallint;
    else
        return (False, (Left.Value + Right.Value));
    end if;
end;

function "*" (Left, Right : SQL_Smallint) return SQL_Smallint is
begin
    if Left.Is_Null or Right.Is_Null then
        return Null_SQL_Smallint;
    else
        return (False, (Left.Value * Right.Value));
    end if;
end;

function "-"(Left, Right : SQL_Smallint) return SQL_Smallint is
begin
    if Left.Is_Null or Right.Is_Null then
        return Null_SQL_Smallint;
    else
        return (False, (Left.Value - Right.Value));
    end if;
end;

function "/"(Left, Right : SQL_Smallint) return SQL_Smallint is
begin
    if Left.Is_Null or Right.Is_Null then

```

```

        return Null_SQL_Smallint;
    else
        return (False, (Left.Value / Right.Value));
    end if;
end;

function "mod" (Left, Right : SQL_Smallint) return SQL_Smallint is
begin
    if Left.Is_Null or Right.Is_Null then
        return Null_SQL_Smallint;
    else
        return (False, (Left.Value mod Right.Value));
    end if;
end;

function "rem" (Left, Right : SQL_Smallint) return SQL_Smallint is
begin
    if Left.Is_Null or Right.Is_Null then
        return Null_SQL_Smallint;
    else
        return (False, (Left.Value rem Right.Value));
    end if;
end;

function "**" (Left : SQL_Smallint; Right: Integer) return SQL_Smallint is
begin
    if Left.Is_Null then
        return Null_SQL_Smallint;
    else
        return (False, (Left.Value ** Right));
    end if;
end;

function IMAGE (Left : SQL_Smallint_Not_Null)
return SQL_Char_Not_Null is
begin
    return to_SQL_Char_Not_Null(
        SQL_Smallint_Not_Null'IMAGE(Left));
end IMAGE;

function IMAGE (Left : SQL_Smallint) return SQL_Char is
begin
    if not Left.Is_Null then
        return to_SQL_Char(
            SQL_Smallint_Not_Null'IMAGE(Left.Value));
    else
        return Null_SQL_Char;
    end if;
end IMAGE;

function VALUE (Left : SQL_Char_Not_Null)
return SQL_Smallint_Not_Null is
begin
    return SQL_Smallint_Not_Null'VALUE(to_String(Left));
end VALUE;

function VALUE (Left : SQL_Char) return SQL_Smallint is
begin
    if Not_Null(Left) then
        return With_Null_Base(
            SQL_Smallint_Not_Null'Value(to_String(Left)));
    else
        return Null_SQL_Smallint;
    end if;
end;

```

```
end if;  
end VALUE;
```

```
-- Logical Operations --  
-- type X type => Boolean_with_Unknown --  
function Equals (Left, Right : SQL_Smallint)  
return Boolean_with_Unknown is
```

```
begin  
if Left.Is_Null or Right.Is_Null then  
return Unknown;  
else  
if (Left.Value = Right.Value) then  
return True;  
else  
return False;  
end if;  
end if;  
end;
```

```
function Not_Equals (Left, Right : SQL_Smallint)  
return Boolean_with_Unknown is
```

```
begin  
if Left.Is_Null or Right.Is_Null then  
return Unknown;  
else  
if (Left.Value = Right.Value) then  
return False;  
else  
return True;  
end if;  
end if;  
end;
```

```
function "<" (Left, Right : SQL_Smallint) return Boolean_with_Unknown is  
begin
```

```
if Left.Is_Null or Right.Is_Null then  
return Unknown;  
else  
if (Left.Value < Right.Value) then  
return True;  
else  
return False;  
end if;  
end if;  
end;
```

```
function ">" (Left, Right : SQL_Smallint) return Boolean_with_Unknown is  
begin
```

```
if Left.Is_Null or Right.Is_Null then  
return Unknown;  
else  
if (Left.Value > Right.Value) then  
return True;  
else  
return False;  
end if;  
end if;  
end;
```

```
function "<=" (Left, Right : SQL_Smallint) return Boolean_with_Unknown is  
begin
```

```
if Left.Is_Null or Right.Is_Null then
```

```

        return Unknown;
    else
        if (Left.Value <= Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end;

function ">=" (Left, Right : SQL_Smallint) return Boolean_with_Unknown is
begin
    if Left.Is_Null or Right.Is_Null then
        return Unknown;
    else
        if (Left.Value >= Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end;

function "=" (Left, Right : SQL_Smallint) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value = Right.Value;
    end if;
end "=";

function "<" (Left, Right : SQL_Smallint) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value < Right.Value;
    end if;
end "<";

function ">" (Left, Right : SQL_Smallint) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value > Right.Value;
    end if;
end ">";

function "<=" (Left, Right : SQL_Smallint) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value <= Right.Value;
    end if;
end "<=";

function ">=" (Left, Right : SQL_Smallint) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value >= Right.Value;
    end if;
end ">=";

```



```

        -- type => boolean --
function Is_Null(Value : SQL_Smallint) return Boolean is
begin
    return Value.Is_Null;
end;

function Not_Null(Value : SQL_Smallint) return Boolean is
begin
    return not Value.Is_Null;
end;

package body SQL_Smallint_Ops is
    function With_Null (Value : Without_Null_Type)
        return With_Null_Type is
    begin
        return With_Null_Base(SQL_Smallint_Not_Null(Value));
    end With_Null;

    function Without_Null (Value : With_Null_Type)
        return Without_Null_Type is
    begin
        return Without_Null_Type(
            SQL_Smallint_Not_Null(Without_Null_Base(Value)));
    end Without_Null;

    procedure assign (Left : in out With_Null_Type;
        Right : in With_Null_Type) is
    begin
        Assign_With_Check(Left, Right,
            SQL_Smallint_Not_Null(Without_Null_Type'FIRST),
            SQL_Smallint_Not_Null(Without_Null_Type'LAST));
    end assign;

end SQL_Smallint_ops;

end SQL_Smallint_Pkg;

```

C.14 SQL_Real_Pkg Specification

```

with SQL_Standard;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
package SQL_Real_Pkg
    is
        type SQL_Real_Not_Null is new SQL_Standard.Real;

        ---- Possibly Null Real ----
        type SQL_Real is limited private;

        function Null_SQL_Real return SQL_Real;
        -- pragma INLINE (Null_SQL_Real);

        -- this pair of functions converts between the
        -- null-bearing and non-null-bearing types
        function Without_Null_Base(Value : SQL_Real)
            return SQL_Real_Not_Null;
        -- pragma INLINE (Without_Null_Base);
        -- With_Null_Base raises Null_Value_Error if the input
        -- value is null
        function With_Null_Base(Value : SQL_Real_Not_Null)
            return SQL_Real;
    end SQL_Real_Pkg;

```

```

-- pragma INLINE (With_Null_Base);

-- this procedure implements range checking
-- note: it is not meant to be used directly
--   by application programmers
-- see the generic package SQL_Real_Ops
-- raises constraint_error if not
--   (First <= Right <= Last)
procedure Assign_with_Check (
    Left : in out SQL_Real; Right : SQL_Real;
    First, Last : SQL_Real Not_Null);
-- pragma INLINE (Assign_with_Check);

-- the following functions implement three valued
-- arithmetic
-- if either input to any of these functions is null
--   the function returns the null value; otherwise
--   they perform the indicated operation
-- these functions raise no exceptions
function "+"(Right : SQL_Real) return SQL_Real;
-- pragma INLINE ("+");
function "-"(Right : SQL_Real) return SQL_Real;
-- pragma INLINE ("-");
function "abs"(Right : SQL_Real) return SQL_Real;
-- pragma INLINE ("abs");
function "+"(Left, Right : SQL_Real) return SQL_Real;
-- pragma INLINE ("+");
function "*" (Left, Right : SQL_Real) return SQL_Real;
-- pragma INLINE ("*");
function "-"(Left, Right : SQL_Real) return SQL_Real;
-- pragma INLINE ("-");
function "/"(Left, Right : SQL_Real) return SQL_Real;
-- pragma INLINE ("/");
function "***"(Left : SQL_Real; Right : Integer) return SQL_Real;
-- pragma INLINE ("***");

-- Logical Operations --
-- type X type => Boolean_with_unknown --
-- these functions implement three valued logic
-- if either input is the null value, the functions
--   return the truth value UNKNOWN; otherwise they
--   perform the indicated comparison.
-- these functions raise no exceptions
function Equals (Left, Right : SQL_Real) return Boolean_with_Unknown;
-- pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Real)
    return Boolean_with_Unknown;
-- pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Real) return Boolean_with_Unknown;
-- pragma INLINE ("<");
function ">" (Left, Right : SQL_Real) return Boolean_with_Unknown;
-- pragma INLINE (">");
function "<=" (Left, Right : SQL_Real) return Boolean_with_Unknown;
-- pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Real) return Boolean_with_Unknown;
-- pragma INLINE (">=");

-- type => boolean --
function Is_Null(Value : SQL_Real) return Boolean;
-- pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Real) return Boolean;
-- pragma INLINE (Not_Null);

```

```

-- These functions of class type => boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.
function "=" (Left, Right : SQL_Real) return Boolean;
-- pragma INLINE ("=");
function "<" (Left, Right : SQL_Real) return Boolean;
-- pragma INLINE ("<");
function ">" (Left, Right : SQL_Real) return Boolean;
-- pragma INLINE (">");
function "<=" (Left, Right : SQL_Real) return Boolean;
-- pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Real) return Boolean;
-- pragma INLINE (">=");

-- this generic is instantiated once for every abstract
-- domain based on the SQL type Real.
-- the three subprogram formal parameters are meant to
-- default to the programs declared above.
-- that is, the package should be instantiated in the
-- scope of the use clause for SQL_Real_Pkg.
-- the two actual types together form the abstract
-- domain.
-- the purpose of the generic is to create functions
-- which convert between the two actual types and a
-- procedure which implements a range constrained
-- assignment for the null-bearing type.
-- the bodies of these subprograms are calls to
-- subprograms declared above and passed as defaults to
-- the generic.
generic
type With_Null_type is limited private;
type Without_null_type is digits <>;
with function With_Null_Base(Value : SQL_Real_Not_Null)
return With_Null_Type is <>;
with function Without_Null_Base(Value : With_Null_Type)
return SQL_Real_Not_Null is <>;
with procedure Assign_with_check (
Left : in out With_Null_Type; Right : With_Null_Type;
First, Last : SQL_Real_Not_Null) is <>;
package SQL_Real_Ops is
function With_Null (Value : Without_Null_type)
return With_Null_type;
-- pragma INLINE (With_Null);
function Without_Null (Value : With_Null_Type)
return Without_Null_type;
-- pragma INLINE (Without_Null);
procedure assign (Left : in out With_Null_Type;
Right : in With_Null_type);
-- pragma INLINE (assign);
end SQL_Real_Ops;

private

type SQL_Real is record
Is_Null: Boolean := true;
Value: SQL_Real_Not_Null;
end record;

end SQL_Real_Pkg;

```

C.15 SQL_Real_Pkg Body

```
with SQL_exceptions;
package Body SQL_Real_pkg is

    Null_Value_Error : exception renames SQL_exceptions.null_value_error;

    function Without_Null_Base(Value : SQL_Real)
    return SQL_Real_Not_Null is
    begin
        if Value.Is_Null then
            raise Null_Value_Error;
        else
            return Value.Value;
        end if;
    end Without_Null_Base;

    function With_Null_Base(Value : SQL_Real_Not_Null)
    return SQL_Real is
    begin
        return(False, Value);
    end With_Null_Base;

    procedure Assign_with_check (
        Left : in out SQL_Real; Right : SQL_Real;
        First, Last : SQL_Real_Not_Null) is
    begin
        if Right.Is_Null then Left.is_Null := True;
        elsif
            (Right.Value < First or else
             Right.Value > Last) then
            raise Constraint_Error;
        else
            Left := Right;
        end if;
    end Assign_With_Check;

    function Null_SQL_Real return SQL_Real is
        Null_Holder : SQL_Real;
    begin
        return (Null_Holder); -- relies on default expression for Is_Null
    end Null_SQL_Real;

    function "+"(Right : SQL_Real) return SQL_Real is
    begin
        return Right;
    end;

    function "-"(Right : SQL_Real) return SQL_Real is
    begin
        return (Right.Is_Null, -(Right.Value));
    end;

    function "abs"(Right : SQL_Real) return SQL_Real is
    begin
        return (Right.Is_Null, abs(Right.Value));
    end;

    function "+"(Left, Right : SQL_Real) return SQL_Real is
    begin
        if Left.Is_Null or Right.Is_Null then
            return Null_SQL_Real;
        else

```

```

        return (False, (Left.Value + Right.Value));
    end if;
end;

function "+"(Left, Right : SQL_Real) return SQL_Real is
begin
    if Left.Is_Null or Right.Is_Null then
        return Null_SQL_Real;
    else
        return (False, (Left.Value * Right.Value));
    end if;
end;

function "-"(Left, Right : SQL_Real) return SQL_Real is
begin
    if Left.Is_Null or Right.Is_Null then
        return Null_SQL_Real;
    else
        return (False, (Left.Value - Right.Value));
    end if;
end;

function "/"(Left, Right : SQL_Real) return SQL_Real is
begin
    if Left.Is_Null or Right.Is_Null then
        return Null_SQL_Real;
    else
        return (False, (Left.Value / Right.Value));
    end if;
end;

function "**"(Left : SQL_Real; Right: Integer) return SQL_Real is
begin
    if Left.Is_Null then
        return Null_SQL_Real;
    else
        return (False, (Left.Value ** Right));
    end if;
end;

-- Logical Operations --
-- type X type => Boolean_with_Unknown --
function Equals (Left, Right : SQL_Real) return Boolean_with_Unknown is
begin
    if Left.Is_Null or Right.Is_Null then
        return Unknown;
    else
        if (Left.Value = Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end;

function Not_Equals (Left, Right : SQL_Real)
    return Boolean_with_Unknown is
begin
    if Left.Is_Null or Right.Is_Null then
        return Unknown;
    else
        if (Left.Value = Right.Value) then

```

```

        return False;
    else
        return True;
    end if;
end if;
end;

function "<" (Left, Right : SQL_Real) return Boolean_with_Unknown is
begin
    if Left.Is_Null or Right.Is_Null then
        return Unknown;
    else
        if (Left.Value < Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end;

function ">" (Left, Right : SQL_Real) return Boolean_with_Unknown is
begin
    if Left.Is_Null or Right.Is_Null then
        return Unknown;
    else
        if (Left.Value > Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end;

function "<=" (Left, Right : SQL_Real) return Boolean_with_Unknown is
begin
    if Left.Is_Null or Right.Is_Null then
        return Unknown;
    else
        if (Left.Value <= Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end;

function ">=" (Left, Right : SQL_Real) return Boolean_with_Unknown is
begin
    if Left.Is_Null or Right.Is_Null then
        return Unknown;
    else
        if (Left.Value >= Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end;

function "=" (Left, Right : SQL_Real) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    end if;
end;

```

```

        else
            return Left.Value = Right.Value;
        end if;
    end "=";
    function "<" (Left, Right : SQL_Real) return Boolean is
    begin
        if Left.Is_Null or else Right.Is_Null then
            return FALSE;
        else
            return Left.Value < Right.Value;
        end if;
    end "<";
    function ">" (Left, Right : SQL_Real) return Boolean is
    begin
        if Left.Is_Null or else Right.Is_Null then
            return FALSE;
        else
            return Left.Value > Right.Value;
        end if;
    end ">";
    function "<=" (Left, Right : SQL_Real) return Boolean is
    begin
        if Left.Is_Null or else Right.Is_Null then
            return FALSE;
        else
            return Left.Value <= Right.Value;
        end if;
    end "<=";
    function ">=" (Left, Right : SQL_Real) return Boolean is
    begin
        if Left.Is_Null or else Right.Is_Null then
            return FALSE;
        else
            return Left.Value >= Right.Value;
        end if;
    end ">=";

    -- type => boolean --
    function Is_Null(Value : SQL_Real) return Boolean is
    begin
        return Value.Is_Null;
    end;

    function Not_Null(Value : SQL_Real) return Boolean is
    begin
        return not Value.Is_Null;
    end;

package body SQL_Real_Ops is
    function With_Null (Value : Without_Null_Type)
        return With_Null_Type is
    begin
        return (With_Null_Base(SQL_Real_Not_Null(Value)));
    end With_Null;

    function Without_Null (Value : With_Null_Type)
        return Without_Null_Type is
    begin
        return (Without_Null_Type(
            SQL_Real_Not_Null'(Without_Null_Base(Value))));
    end Without_Null;

    procedure assign (Left : in out With_Null_Type;

```

```

                Right : in With_null_type) is
begin
    Assign_With_Check(Left, Right,
        SQL_Real_Not_Null(Without_Null_Type'FIRST),
        SQL_Real_Not_Null(Without_Null_Type'LAST));
end assign;

end SQL_Real_Ops;

end SQL_Real_Pkg;

```

C.16 SQL_Double_Precision_Pkg Specification

```

with SQL_Standard;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
package SQL_Double_Precision_Pkg
    is
    type SQL_Double_Precision_Not_Null is new SQL_Standard.Double_Precision;

        ---- Possibly Null Double_Precision ----
    type SQL_Double_Precision is limited private;

    function Null_SQL_Double_Precision return SQL_Double_Precision;
    -- pragma INLINE (Null_SQL_Double_Precision);

    -- this pair of functions converts between the
    -- null-bearing and non-null-bearing types.
    function Without_Null_Base(Value : SQL_Double_Precision)
        return SQL_Double_Precision_Not_Null;
    -- pragma INLINE (Without_Null_Base);
    -- With_Null_Base raises Null_Value_Error if the input
    -- value is null
    function With_Null_Base(Value : SQL_Double_Precision_Not_Null)
        return SQL_Double_Precision;
    -- pragma INLINE (With_Null_Base);

    -- this procedure implements range checking
    -- note: it is not meant to be used directly
    -- by application programmers
    -- see the generic package SQL_Double_Precision_Op
    -- raises constraint_error if not
    -- (First <= Right <= Last)
    procedure Assign_with_Check (
        Left : in out SQL_Double_Precision;
        Right : SQL_Double_Precision;
        First, Last : SQL_Double_Precision_Not_Null);
    -- pragma INLINE (Assign_with_Check);

    -- the following functions implement three valued
    -- arithmetic
    -- if either input to any of these functions is null
    -- the function returns the null value; otherwise they
    -- perform the indicated operation
    -- these functions raise no exceptions
    function "+"(Right : SQL_Double_Precision) return SQL_Double_Precision;
    -- pragma INLINE ("+");
    function "-"(Right : SQL_Double_Precision) return SQL_Double_Precision;
    -- pragma INLINE ("-");
    function "abs"(Right : SQL_Double_Precision) return SQL_Double_Precision;
    -- pragma INLINE ("abs");

```



```

function "+"(Left, Right : SQL_Double_Precision)
    return SQL_Double_Precision;
-- pragma INLINE ("+");
function "*" (Left, Right : SQL_Double_Precision)
    return SQL_Double_Precision;
-- pragma INLINE ("*");
function "-"(Left, Right : SQL_Double_Precision)
    return SQL_Double_Precision;
-- pragma INLINE ("-");
function "/"(Left, Right : SQL_Double_Precision)
    return SQL_Double_Precision;
-- pragma INLINE ("/");
function "***"(Left : SQL_Double_Precision; Right : Integer)
    return SQL_Double_Precision;
-- pragma INLINE ("***");

-- Logical Operations --
-- type X type => Boolean_with_unknown --
-- these functions implement three valued logic
-- if either input is the null value, the functions
-- return the truth value UNKNOWN; otherwise they
-- perform the indicated comparison.
-- these functions raise no exceptions
function Equals (Left, Right : SQL_Double_Precision)
    return Boolean_with_Unknown;
-- pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Double_Precision)
    return Boolean_with_Unknown;
-- pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Double_Precision)
    return Boolean_with_Unknown;
-- pragma INLINE ("<");
function ">" (Left, Right : SQL_Double_Precision)
    return Boolean_with_Unknown;
-- pragma INLINE (">");
function "<=" (Left, Right : SQL_Double_Precision)
    return Boolean_with_Unknown;
-- pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Double_Precision)
    return Boolean_with_Unknown;
-- pragma INLINE (">=");

-- type => boolean --
function Is_Null(Value : SQL_Double_Precision) return Boolean;
-- pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Double_Precision) return Boolean;
-- pragma INLINE (Not_Null);

-- These functions of class type => boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.
function "=" (Left, Right : SQL_Double_Precision) return Boolean;
function "<" (Left, Right : SQL_Double_Precision) return Boolean;
function ">" (Left, Right : SQL_Double_Precision) return Boolean;
function "<=" (Left, Right : SQL_Double_Precision) return Boolean;
function ">=" (Left, Right : SQL_Double_Precision) return Boolean;

-- this generic is instantiated once for every abstract
-- domain based on the SQL type Double_Precision.
-- the three subprogram formal parameters are meant to
-- default to the programs declared above.

```

```

-- that is, the package should be instantiated in the
-- scope of the use clause for
-- SQL_Double_Precision_Pkg.
-- the two actual types together form the abstract
-- domain.
-- the purpose of the generic is to create functions
-- which convert between the two actual types and a
-- procedure which implements a range constrained
-- assignment for the null-bearing type.
-- the bodies of these subprograms are calls to
-- subprograms declared above and passed as defaults
-- to the generic.
generic
type With_Null_type is limited private;
type Without_null_type is digits <>;
with function With_Null_Base(Value : SQL_Double_Precision_Not_Null)
return With_Null_Type is <>;
with function Without_Null_Base(Value : With_Null_Type)
return SQL_Double_Precision_Not_Null is <>;
with procedure Assign_with_check (
Left : in out With_Null_Type; Right : With_Null_Type;
First, Last : SQL_Double_Precision_Not_Null) is <>;
package SQL_Double_Precision_Ops is
function With_Null (Value : Without_Null_type)
return With_Null_type;
-- pragma INLINE (With_Null);
function Without_Null (Value : With_Null_Type)
return Without_Null_type;
-- pragma INLINE (Without_Null);
procedure assign (Left : in out With_null_Type;
Right : in With_null_type);
-- pragma INLINE (assign);
end SQL_Double_Precision_Ops;

private

type SQL_Double_Precision is record
Is_Null: Boolean := true;
Value: SQL_Double_Precision_Not_Null;
end record;

end SQL_Double_Precision_Pkg;

```

C.17 SQL_Double_Precision_Pkg Body

```

with SQL_exceptions;
package body SQL_Double_Precision_pkg is

Null_Value_Error : exception renames SQL_exceptions.null_value_error;

function Without_Null_Base(Value : SQL_Double_Precision)
return SQL_Double_Precision_Not_Null is
begin
if Value.Is_Null then
raise Null_Value_Error;
else
return Value.Value;
end if;
end Without_Null_Base;

```

```

function With_Null_Base(Value : SQL_Double_Precision_Not_Null)
  return SQL_Double_Precision is
begin
  return(False, Value);
end With_Null_Base;

procedure Assign_with_check (
  Left : in out SQL_Double_Precision; Right : SQL_Double_Precision;
  First, Last : SQL_Double_Precision_Not_Null) is
begin
  if Right.Is_Null then Left.Is_Null := True;
  elsif
    (Right.Value < First or else
     Right.Value > Last) then
    raise Constraint_Error;
  else
    Left := Right;
  end if;
end Assign_With_Check;

function Null_SQL_Double_Precision return SQL_Double_Precision is
  Null_Holder : SQL_Double_Precision;
begin
  return (Null_Holder); -- relies on default expression for Is_Null
end Null_SQL_Double_Precision;

function "+"(Right : SQL_Double_Precision)
  return SQL_Double_Precision is
begin
  return Right;
end;

function "-"(Right : SQL_Double_Precision)
  return SQL_Double_Precision is
begin
  return (Right.Is_null, -(Right.Value));
end;

function "abs"(Right : SQL_Double_Precision)
  return SQL_Double_Precision is
begin
  return (Right.Is_null, abs(Right.Value));
end;

function "+"(Left, Right : SQL_Double_Precision)
  return SQL_Double_Precision is
begin
  if Left.Is_Null or Right.Is_Null then
    return Null_SQL_Double_Precision;
  else
    return (False, (Left.Value + Right.Value));
  end if;
end;

function "*" (Left, Right : SQL_Double_Precision)
  return SQL_Double_Precision is
begin
  if Left.Is_Null or Right.Is_Null then
    return Null_SQL_Double_Precision;
  else
    return (False, (Left.Value * Right.Value));
  end if;
end;

```

```

function "-" (Left, Right : SQL_Double_Precision)
  return SQL_Double_Precision is
begin
  if Left.Is_Null or Right.Is_Null then
    return Null_SQL_Double_Precision;
  else
    return (False, (Left.Value - Right.Value));
  end if;
end;

function "/" (Left, Right : SQL_Double_Precision)
  return SQL_Double_Precision is
begin
  if Left.Is_Null or Right.Is_Null then
    return Null_SQL_Double_Precision;
  else
    return (False, (Left.Value / Right.Value));
  end if;
end;

function "*" (Left : SQL_Double_Precision; Right: Integer)
  return SQL_Double_Precision is
begin
  if Left.Is_Null then
    return Null_SQL_Double_Precision;
  else
    return (False, (Left.Value ** Right));
  end if;
end;

-- Logical Operations --
-- type X type => Boolean_with_unknown --
function Equals (Left, Right : SQL_Double_Precision)
  return Boolean_with_Unknown is
begin
  if Left.Is_Null or Right.Is_Null then
    return Unknown;
  else
    if (Left.Value = Right.Value) then
      return True;
    else
      return False;
    end if;
  end if;
end;

function Not_Equals (Left, Right : SQL_Double_Precision)
  return Boolean_with_Unknown is
begin
  if Left.Is_Null or Right.Is_Null then
    return Unknown;
  else
    if (Left.Value = Right.Value) then
      return False;
    else
      return True;
    end if;
  end if;
end;

function "<" (Left, Right : SQL_Double_Precision)
  return Boolean_with_Unknown is

```

```

begin
  if Left.Is_Null or Right.Is_Null then
    return Unknown;
  else
    if (Left.Value < Right.Value) then
      return True;
    else
      return False;
    end if;
  end if;
end;

function ">" (Left, Right : SQL_Double_Precision)
  return Boolean_with_Unknown is
begin
  if Left.Is_Null or Right.Is_Null then
    return Unknown;
  else
    if (Left.Value > Right.Value) then
      return True;
    else
      return False;
    end if;
  end if;
end;

function "<=" (Left, Right : SQL_Double_Precision)
  return Boolean_with_Unknown is
begin
  if Left.Is_Null or Right.Is_Null then
    return Unknown;
  else
    if (Left.Value <= Right.Value) then
      return True;
    else
      return False;
    end if;
  end if;
end;

function ">=" (Left, Right : SQL_Double_Precision)
  return Boolean_with_Unknown is
begin
  if Left.Is_Null or Right.Is_Null then
    return Unknown;
  else
    if (Left.Value >= Right.Value) then
      return True;
    else
      return False;
    end if;
  end if;
end;

-- type => boolean --
function Is_Null(Value : SQL_Double_Precision) return Boolean is
begin
  return Value.Is_Null;
end;

function Not_Null(Value : SQL_Double_Precision) return Boolean is
begin

```

```

        return not Value.Is_Null;
end;

function "=" (Left, Right : SQL_Double_Precision) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value = Right.Value;
    end if;
end "=";
function "<" (Left, Right : SQL_Double_Precision) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value < Right.Value;
    end if;
end "<";
function ">" (Left, Right : SQL_Double_Precision) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value > Right.Value;
    end if;
end ">";
function "<=" (Left, Right : SQL_Double_Precision) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value <= Right.Value;
    end if;
end "<=";
function ">=" (Left, Right : SQL_Double_Precision) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        return Left.Value >= Right.Value;
    end if;
end ">=";

package body SQL_Double_Precision_Ops is
    function With_Null (Value : Without_Null_Type)
        return With_Null_Type is
    begin
        return (With_Null_Base(SQL_Double_Precision_Not_Null(Value)));
    end With_Null;

    function Without_Null (Value : With_Null_Type)
        return Without_Null_Type is
    begin
        return (Without_Null_Type(
            SQL_Double_Precision_Not_Null(Without_Null_Base(Value))));
    end Without_Null;

    procedure assign (Left : in out With_Null_Type;
        Right : in With_Null_Type) is
    begin
        Assign_With_Check(Left, Right,
            SQL_Double_Precision_Not_Null(Without_Null_Type'FIRST));
    end assign;

```

```

        SQL_Double_Precision_Not_Null(Without_Null_Type'LAST));
    end assign;

end SQL_Double_Precision_Ops;

end SQL_Double_Precision_Pkg;

```

C.18 SQL_Decimal_Pkg Specification

```

with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Int_Pkg; use SQL_Int_Pkg;
with SQL_Char_Pkg; use SQL_Char_Pkg;
with SQL_Double_Precision_Pkg; use SQL_Double_Precision_Pkg;
package SQL_Decimal_Pkg is

    -- MAX_DIGITS is implementation defined
    -- It represents the maximum number of digits that can be
    -- stored in the underlying hardware's representation of
    -- a BCD number
    MAX_DIGITS : constant integer := 31;

    subtype decimal_digits is natural range 0..MAX_DIGITS;

    type SQL_Decimal_Not_Null(scale : decimal_digits := 0) is limited private;
    type SQL_Decimal(scale : decimal_digits) is limited private;

    subtype Numeric_Character is Character range '0'..'9';
    type Numeric_String is array (decimal_digits range <>) of Numeric_Character;
    type Sign_Character is ('+', '-');

    -- the following type is used for purposes of creating generic
    -- assign and is_in functions....DO NOT USE THIS TYPE to
    -- create the abstract domains.....
    type SQL_Decimal_Not_Null2(scale : decimal_digits := 0) is limited private;

    function To_SQL_Decimal_Not_Null (Value : SQL_Decimal_Not_Null2)
        return SQL_Decimal_Not_Null;
    function To_SQL_Decimal (Value : SQL_Decimal_Not_Null2)
        return SQL_Decimal;
    function To_SQL_Decimal_Not_Null2 (Value : SQL_Decimal_Not_Null)
        return SQL_Decimal_Not_Null2;
    function To_SQL_Decimal_Not_Null2 (Value : SQL_Decimal)
        return SQL_Decimal_Not_Null2;
    -- pragma INLINE(To_SQL_Decimal_Not_Null2);

    -- this function returns a null value of the SQL_Decimal type
    function Null_SQL_Decimal return SQL_Decimal;
    -- pragma INLINE(Null_SQL_Decimal);

    -- The following functions shift the value of the object
    -- without changing the scale. Effectively, the operation
    -- multiplies the value in the object by 10**Scale.
    -- The following functions raise Constraint_Error if the left
    -- shift causes a loss of significant digits
    function Shift (Value : SQL_Decimal_Not_Null;
        Scale : integer) return SQL_Decimal_Not_Null;
    function Shift (Value : SQL_Decimal;
        Scale : integer) return SQL_Decimal;
    -- pragma INLINE(Shift);

    -- The following functions return objects with the appropriate

```

```

-- values
function Zero return SQL_Decimal_Not_Null;
function Zero return SQL_Decimal;
-- pragma INLINE(Zero);
function One return SQL_Decimal_Not_Null;
function One return SQL_Decimal;
-- pragma INLINE(One);

-- The following Assignment procedure is provided for the
-- SQL_Decimal_Not_Null type:
-- The following Assignment procedure raises Constraint_Error
-- if the value of Right does not fall within the range
-- of lower..upper
procedure Assign_With_Check (Left : in out SQL_Decimal_Not_Null;
                             Right : SQL_Decimal_Not_Null;
                             Lower, Upper : SQL_Decimal_Not_Null2);

-- The following Assign_with_check procedure will be used
-- in the generic Assign produced in SQL_Decimal_Ops
-- this procedure raises the Constraint_Error exception if
-- the "Right" input parameter falls outside the range
-- defined by Lower..Upper
procedure Assign_With_Check
    (Left : in out SQL_Decimal;
     Right : SQL_Decimal;
     Lower, Upper : SQL_Decimal_Not_Null2);
-- pragma INLINE(Assign_with_check);

-- The following comparison operators are provided:

function "=" (Left, Right : SQL_Decimal_Not_Null) return boolean;
function "=" (Left, Right : SQL_Decimal) return boolean;
-- pragma INLINE("=");
function Equals (Left, Right : SQL_Decimal) return Boolean_With_Unknown;
-- pragma INLINE(Equals);
function Not_Equals (Left, Right : SQL_Decimal) return Boolean_With_Unknown;
-- pragma INLINE(Not_Equals);
function "<" (Left, Right : SQL_Decimal_Not_Null) return boolean;
function "<" (Left, Right : SQL_Decimal) return boolean;
function "<" (Left, Right : SQL_Decimal) return Boolean_With_Unknown;
-- pragma INLINE("<");
function ">" (Left, Right : SQL_Decimal_Not_Null) return boolean;
function ">" (Left, Right : SQL_Decimal) return boolean;
function ">" (Left, Right : SQL_Decimal) return Boolean_With_Unknown;
-- pragma INLINE(">");
function "<=" (Left, Right : SQL_Decimal_Not_Null) return boolean;
function "<=" (Left, Right : SQL_Decimal) return boolean;
function "<=" (Left, Right : SQL_Decimal) return Boolean_With_Unknown;
-- pragma INLINE("<=");
function ">=" (Left, Right : SQL_Decimal_Not_Null) return boolean;
function ">=" (Left, Right : SQL_Decimal) return boolean;
function ">=" (Left, Right : SQL_Decimal) return Boolean_With_Unknown;
-- pragma INLINE(">=");

-- the following functions are membership tests
-- the value of the object is tested to see if
-- if it falls within the range of Lower..Upper
function Is_In_Base (Right : SQL_Decimal_Not_Null;
                    Lower, Upper : SQL_Decimal_Not_Null2)
    return boolean;
function Is_In_Base (Right : SQL_Decimal;
                    Lower, Upper : SQL_Decimal_Not_Null2)
    return boolean;

```



```

-- pragma INLINE (Is_In_Base);

function Is_Null(Value : SQL_Decimal) return boolean;
-- pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Decimal) return boolean;
-- pragma INLINE (Not_Null);

-- The following unary arithmetic operators are provided:
function "+" (Right : SQL_Decimal_Not_Null)
    return SQL_Decimal_Not_Null;
function "+" (Right : SQL_Decimal) return SQL_Decimal;
function "-" (Right : SQL_Decimal_Not_Null)
    return SQL_Decimal_Not_Null;
function "-" (Right : SQL_Decimal) return SQL_Decimal;
function "abs" (Right : SQL_Decimal_Not_Null)
    return SQL_Decimal_Not_Null;
function "abs" (Right : SQL_Decimal) return SQL_Decimal;
-- pragma INLINE("abs");

-- The following binary arithmetic operators are provided:

-- The "+" and "-" functions return a result with a scale of
-- max(Left.scale, Right.scale)
-- If the operation produces a result that is too large to
-- be represented in an object that has this scale, a
-- Constraint_Error will be raised
function "+" (Left, Right : SQL_Decimal_Not_Null)
    return SQL_Decimal_Not_Null;
function "+" (Left, Right : SQL_Decimal) return SQL_Decimal;
-- pragma INLINE("+");
function "-" (Left, Right : SQL_Decimal_Not_Null)
    return SQL_Decimal_Not_Null;
function "-" (Left, Right : SQL_Decimal) return SQL_Decimal;
-- pragma INLINE("-");
-- The "*" function returns a result with the scale
-- Left.scale + Right.scale
-- If the result is too large to be represented in an object
-- that has this scale, Constraint_Error will be raised
function "*" (Left, Right : SQL_Decimal_Not_Null)
    return SQL_Decimal_Not_Null;
function "*" (Left, Right : SQL_Decimal) return SQL_Decimal;
-- The "/" function returns a result with as much scale as
-- possible, given the nature of the result
-- If the result is too large to be represented in the
-- underlying hardware or in an object with no scale,
-- or if an attempt is made to divide by zero, the
-- Constraint_Error exception will be raised
function "/" (Left, Right : SQL_Decimal_Not_Null)
    return SQL_Decimal_Not_Null;
function "/" (Left, Right : SQL_Decimal) return SQL_Decimal;

-- The following mixed mode operators are provided:
function "*" (Left : SQL_Decimal_Not_Null; Right : SQL_Int_Not_Null)
    return SQL_Decimal_Not_Null;
function "*" (Left : SQL_Decimal; Right : SQL_Int_Not_Null)
    return SQL_Decimal;
function "*" (Left : SQL_Decimal; Right : SQL_Int)
    return SQL_Decimal;
function "*" (Left : SQL_Int_Not_Null; Right : SQL_Decimal_Not_Null)
    return SQL_Decimal_Not_Null;
function "*" (Left : SQL_Int_Not_Null; Right : SQL_Decimal)
    return SQL_Decimal;
function "*" (Left : SQL_Int; Right : SQL_Decimal)

```

```

    return SQL_Decimal;
-- pragma INLINE("*");
function "/" (Left : SQL_Decimal_Not_Null; Right : SQL_Int_Not_Null)
    return SQL_Decimal_Not_Null;
function "/" (Left : SQL_Decimal; Right : SQL_Int_Not_Null)
    return SQL_Decimal;
function "/" (Left : SQL_Decimal; Right : SQL_Int)
    return SQL_Decimal;
-- pragma INLINE("/");

-- The following functions convert to SQL_Decimal_Not_Null;
function To_SQL_Decimal_Not_Null (Right : SQL_Int_Not_Null)
    return SQL_Decimal_Not_Null;
-- the following function raises Constraint_Error
-- if the SQL_Double_Precision_Not_Null value is too large
-- to be represented in BCD format
function To_SQL_Decimal_Not_Null (Right : SQL_Double_Precision_Not_Null)
    return SQL_Decimal_Not_Null;
-- the following function raises Constraint_Error
-- if there are more than MAX_DIGITS number of digits;
-- if there are two or more decimal points;
-- if there are two or more sign designations;
-- if there exists a character other than '0'..'9' or '.'
-- or '+', '-', ' ' for the sign
-- if the order of the characters is anything other than
-- sign designation followed by the number
function To_SQL_Decimal_Not_Null (Right : SQL_Char_Not_Null)
    return SQL_Decimal_Not_Null;
-- pragma INLINE(To_SQL_Decimal_Not_Null);

-- The following functions convert to SQL_Decimal;
function To_SQL_Decimal (Right : SQL_Int_Not_Null) return SQL_Decimal;
function To_SQL_Decimal (Right : SQL_Int) return SQL_Decimal;
-- the following two functions raise Constraint_Error
-- if the SQL_Double_Precision_Not_Null value is too large
-- to be represented in BCD format
function To_SQL_Decimal (Right : SQL_Double_Precision_Not_Null)
    return SQL_Decimal;
function To_SQL_Decimal (Right : SQL_Double_Precision) return SQL_Decimal;
-- the following two functions raise Constraint_Error
-- if there are more than MAX_DIGITS number of digits;
-- if there are two or more decimal points;
-- if there are two or more sign designations;
-- if there exists a character other than '0'..'9' or '.'
-- or '+', '-', ' ' for the sign
-- if the order of the characters is anything other than
-- sign designation followed by the number
function To_SQL_Decimal (Right : SQL_Char_Not_Null) return SQL_Decimal;
function To_SQL_Decimal (Right : SQL_Char) return SQL_Decimal;
-- pragma INLINE(To_SQL_Decimal);

-- The following functions convert from Decimal to Integer
function To_SQL_Int_Not_Null (Right : SQL_Decimal_Not_Null)
    return SQL_Int_Not_Null;
function To_SQL_Int_Not_Null (Right : SQL_Decimal)
    return SQL_Int_Not_Null;
-- pragma INLINE(To_SQL_Int_Not_Null);
function To_SQL_Int (Right : SQL_Decimal) return SQL_Int;
-- pragma INLINE(To_SQL_Int);

-- The following functions convert from Decimal to Float:
function To_SQL_Double_Precision_Not_Null (Right : SQL_Decimal_Not_Null)
    return SQL_Double_Precision_Not_Null;

```

```

function To_SQL_Double_Precision_Not_Null (Right : SQL_Decimal)
    return SQL_Double_Precision_Not_Null;
-- pragma INLINE(To_SQL_Double_Precision_Not_Null);
function To_SQL_Double_Precision (Right : SQL_Decimal)
    return SQL_Double_Precision;
-- pragma INLINE(To_SQL_Double_Precision);

-- The following functions convert from Decimal to String:
function To_String (Right : SQL_Decimal_Not_Null) return string;
function To_String (Right : SQL_Decimal) return string;
-- pragma INLINE(To_String);
function To_SQL_Char_Not_Null (Right : SQL_Decimal_Not_Null)
    return SQL_Char_Not_Null;
function To_SQL_Char_Not_Null (Right : SQL_Decimal)
    return SQL_Char_Not_Null;
-- pragma INLINE(To_SQL_Char_Not_Null);
function To_SQL_Char (Right : SQL_Decimal) return SQL_Char;
-- pragma INLINE(To_SQL_Char);

-- the following functions return the length of the string
-- value returned by the "To_String" function
function Width (Right : SQL_Decimal_Not_Null) return integer;
-- The following function raises the Null_Value_Error exception
-- on the null input
function Width (Right : SQL_Decimal) return integer;
-- pragma INLINE(Width);

-- The following functions implement some of the Ada Attributes
-- of the BCD type

-- The number of BCD digits before the decimal point for the
-- type of the given object:
function Integral_Digits (Right : SQL_Decimal_Not_Null) return decimal_digits;
function Integral_Digits (Right : SQL_Decimal) return decimal_digits;
-- pragma INLINE(Integral_Digits);

-- The number of BCD digits after the decimal point for the
-- type of the given object:
function Scale (Right : SQL_Decimal_Not_Null) return decimal_digits;
function Scale (Right : SQL_Decimal) return decimal_digits;
-- pragma INLINE(Scale);

-- The actual number of BCD digits before the decimal point for
-- a given object of a given type:
function Fore (Right : SQL_Decimal_Not_Null) return positive;
-- The following function raises the Null_Value_Error on the null input
function Fore (Right : SQL_Decimal) return positive;
-- pragma INLINE(Fore);

-- The number of BCD digits after the decimal point for a
-- given object of a given type:
function Aft (Right : SQL_Decimal_Not_Null) return positive;
-- The following function raises the Null_Value_Error on the null input
function Aft (Right : SQL_Decimal) return positive;
-- pragma INLINE(Aft);

function Machine_Rounds (Right : SQL_Decimal_Not_Null) return boolean;
function Machine_Rounds (Right : SQL_Decimal) return boolean;
-- pragma INLINE(Machine_Rounds);

function Machine_Overflows (Right : SQL_Decimal_Not_Null) return boolean;
function Machine_Overflows (Right : SQL_Decimal) return boolean;
-- pragma INLINE(Machine_Overflows);

```

```

generic
  type With_Null_Type(scale : decimal_digits) is limited private;
  type Without_Null_Type(scale : decimal_digits) is limited private;
  in_scale      : decimal_digits := 0;
  first_sign    : Sign_Character := '-';
  first_integral : Numeric_String :=
    (1..decimal_digits'last-in_scale => '9');
  first_fractional : Numeric_String :=
    (1..in_scale => '9');
  last_sign     : Sign_Character := '+';
  last_integral : Numeric_String :=
    (1..decimal_digits'last-in_scale => '9');
  last_fractional : Numeric_String :=
    (1..in_scale => '9');
  with function Is_In_Base (Right : Without_Null_Type;
    Lower, Upper : SQL_Decimal_Not_Null2)
    return boolean is <>;
  with function Is_In_Base (Right : With_Null_Type;
    Lower, Upper : SQL_Decimal_Not_Null2)
    return boolean is <>;
  with procedure Assign_with_check
    (Left : in out Without_Null_Type;
    Right : Without_Null_Type;
    Lower, Upper : SQL_Decimal_Not_Null2)
    is <>;
  with procedure Assign_with_check
    (Left : in out With_Null_Type;
    Right : With_Null_Type;
    Lower, Upper : SQL_Decimal_Not_Null2)
    is <>;
  with function To_SQL_Decimal_Not_Null2 (Value : Without_Null_Type)
    return SQL_Decimal_Not_Null2 is <>;
  with function To_SQL_Decimal_Not_Null2 (Value : With_Null_Type)
    return SQL_Decimal_Not_Null2 is <>;
  with function To_SQL_Decimal_Not_Null (Value : SQL_Decimal_Not_Null2)
    return Without_Null_Type is <>;
  with function To_SQL_Decimal (Value : SQL_Decimal_Not_Null2)
    return With_Null_Type is <>;
package SQL_Decimal_Ops is
  procedure Assign (Left : in out Without_Null_Type;
    Right : Without_Null_Type);
  procedure Assign (Left : in out With_Null_Type;
    Right : With_Null_Type);
  -- pragma INLINE(Assign);
  function Is_In(Right : Without_Null_Type)
    return boolean;
  function Is_In(Right : With_Null_Type)
    return boolean;
  -- pragma INLINE(Is_In);
  function With_Null (Value : With_Null_Type)
    return With_Null_Type;
  -- pragma INLINE(With_Null);
  function Without_Null (Value : With_Null_Type)
    return Without_Null_Type;
  -- pragma INLINE(Without_Null_Type);
end SQL_Decimal_Ops;

private

  -- The requirement here is to provide
  -- at least enough space for the machine representation of the
  -- SQL_Decimal_Not_Null operands.

```

```

-- type Digit is picked to be an integer type with a range
--   that will force the Ada compiler to pick a
--   pre-defined integer type from package Standard.

type Digit is range -(2**7)..(2**7)-1;

-- the following object is declared so that the true size
--   (in actual number of bits allocated) is assigned to the
--   "size" object, rather than the number of bits used of
--   those which are allocated.  In other words, using 'size
--   on the type Digit yields 4 bits (number bits used),
--   whereas using the 'size on "object" (of type Digit) yields
--   8 bits (number bits allocated)

object : Digit;

-- size is the number of bits used by each object of type Digit
-- it is used in the calculation of MAX_SIZE (below)

size : constant integer := object'size;

-- MAX_SIZE is the number of array positions needed for the
--   Max_Decimal type below
-- since each BCD digit can fit into 4 bits of storage, the
--   total number of bits can be calculated by MAX_DIGITS * 4;
-- this result is divided by the number of bits that an object
--   of type Digit will comprise, which yields the number of
--   array positions needed for the BCD number
-- the result is incremented by one to accommodate the sign

MAX_SIZE : constant integer := ((4 * (MAX_DIGITS)) / size) + 1;

-- Max_Decimal is the array type definition used by the
--   SQL_Decimal_Not_Null type definition (below) to allocate maximum
--   storage for its BCD value

type Max_Decimal is array (1..MAX_SIZE) of Digit;

-- SQL_Decimal_Not_Null is the Ada BCD type.  It is comprised of a BCD
--   value which resides in an object which reserves maximum
--   space for BCD values, and a scale which indicates how
--   many digits exist to the right of the decimal point in the
--   BCD value

type SQL_Decimal_Not_Null (scale : decimal_digits := 0) is record
  Value : Max_Decimal;
end record;

type SQL_Decimal_Not_Null2 (scale : decimal_digits := 0) is record
  Value : Max_Decimal;
end record;

type SQL_Decimal(scale : decimal_digits) is record
  Is_Null : boolean := true;
  Value : SQL_Decimal_Not_Null(scale);
end record;

end SQL_Decimal_Pkg;

```

C.19 SQL_Decimal_Pkg Body

```
with text_io; use text_io;
with unchecked_conversion;
with SQL_Exceptions;
with SQL_Standard;
package body SQL_Decimal_Pkg is

    -- the following type is used to convert all other integer
    -- types to the underlying hardware integer representation
    -- used by the computer to convert between integers
    -- and packed decimal numbers

    type BCD_Int_Type is range -(2**31)..(2**31)-1;

    Null_Value_Error : exception renames SQL_Exceptions.null_value_error;
    package fio is new float_io(float); use fio;
    package SQL_DPNN_io is new float_io(SQL_Double_Precision_Not_Null);
    use SQL_DPNN_io;
    use SQL_Standard.Character_Set;

    -- interfaced assembler routines
    -----

    -- this procedure converts the integer in Right to a BCD value
    procedure integer_to_decimal (Value : in out Max_Decimal;
                                  Right : BCD_Int_Type);
    pragma interface (assembler, integer_to_decimal);
    pragma import_procedure (integer_to_decimal, "I2D",
                              (Max_Decimal, BCD_Int_Type), Reference);

    -- this procedure converts the BCD value in Right to an integer
    procedure decimal_to_integer (Value : in out BCD_Int_Type;
                                   Right : Max_Decimal;
                                   error : in out boolean);
    pragma interface (assembler, decimal_to_integer);
    pragma import_procedure (decimal_to_integer, "D2I",
                              (BCD_Int_Type, Max_Decimal, boolean), Reference);

    -- this procedure converts a string representation of a BCD value
    -- into a BCD value
    procedure numeric_string_to_decimal (Value : in out Max_Decimal;
                                           Right : SQL_Char_Not_Null);
    pragma interface (assembler, numeric_string_to_decimal);
    pragma import_procedure (numeric_string_to_decimal, "NS2D",
                              (Max_Decimal, SQL_Char_Not_Null),
                              Reference);

    -- this procedure converts a BCD value to a string representation
    -- of that value
    procedure decimal_to_numeric_string (Value : in out SQL_Char_Not_Null;
                                           Right : Max_Decimal);
    pragma interface (assembler, decimal_to_numeric_string);
    pragma import_procedure (decimal_to_numeric_string, "D2NS",
                              (SQL_Char_Not_Null, Max_Decimal), Reference);

    -- this procedure returns the number of leading zeroes in the
    -- first "integ" digits of the BCD value
    procedure leading_zeroes (Value : Max_Decimal;
                               integ : integer;
                               digs : in out integer);
    pragma interface (assembler, leading_zeroes);
    pragma import_procedure (leading_zeroes, "LZ",
```

```

                                (Max_Decimal, integer, integer),
                                Reference);

-- this procedure returns the number of trailing zeroes in the
-- last "scal" digits of the BCD value
procedure trailing_zeroes (Value : Max_Decimal;
                           scal : decimal_digits;
                           digs : in out integer);
pragma interface (assembler, trailing_zeroes);
pragma import_procedure (trailing_zeroes, "TZ",
                        (Max_Decimal, decimal_digits,
                         integer), Reference);

-- this procedure interprets the sign of the BCD value, and
-- negates it
procedure inverse (Value : in out Max_Decimal;
                  Right : Max_Decimal);
pragma interface (assembler, inverse);
pragma import_procedure (inverse, "INV",
                        (Max_Decimal, Max_Decimal),
                        Reference);

-- this procedure returns the absolute value of the BCD value
procedure absv (Value : in out Max_Decimal;
               Right : Max_Decimal);
pragma interface (assembler, absv);
pragma import_procedure (absv, "ABSV",
                        (Max_Decimal, Max_Decimal),
                        Reference);

-- this procedure shifts the input value by "scale" powers of 10
-- if "scale" is positive, the shift is left; else the shift is
-- right
procedure shft (Result : out Max_Decimal;
               Value : Max_Decimal;
               scale : integer;
               error : in out boolean);
pragma interface (assembler, shft);
pragma import_procedure (shft, "SHFT",
                        (Max_Decimal, Max_Decimal, integer, boolean),
                        Reference);

-- this procedure determines if Left and Right are equal
procedure equal (Left, Right : Max_Decimal;
                result : in out boolean);
pragma interface (assembler, equal);
pragma import_procedure (equal, "EQ",
                        (Max_Decimal, Max_Decimal, boolean),
                        Reference);

-- this procedure determines if Left is < Right
procedure less_than (Left, Right : Max_Decimal;
                    result : in out boolean);
pragma interface (assembler, less_than);
pragma import_procedure (less_than, "LT",
                        (Max_Decimal, Max_Decimal, boolean),
                        Reference);

-- this procedure determines if Left > Right
procedure greater_than (Left, Right : Max_Decimal;
                       result : in out boolean);
pragma interface (assembler, greater_than);
pragma import_procedure (greater_than, "GT",

```

```

        (Max_Decimal, Max_Decimal, boolean),
        Reference);

-- this procedure determines if Left <= Right
procedure less_than_equal (Left, Right : Max_Decimal;
                           result      : in out boolean);
pragma interface (assembler, less_than_equal);
pragma import_procedure (less_than_equal, "LEQ",
                        (Max_Decimal, Max_Decimal, boolean),
                        Reference);

-- this procedure determines if Left >= Right
procedure greater_than_equal (Left, Right : Max_Decimal;
                              result      : in out boolean);
pragma interface (assembler, greater_than_equal);
pragma import_procedure (greater_than_equal, "GEQ",
                        (Max_Decimal, Max_Decimal, boolean),
                        Reference);

-- this procedure adds Left and Right, and stores the result
-- in Result
-- the "error" boolean is set to true on overflow
procedure add (Result : in out Max_Decimal;
              Left, Right : Max_Decimal;
              error : in out boolean);
pragma interface (assembler, add);
pragma import_procedure (add, "ADD",
                        (Max_Decimal, Max_Decimal,
                         Max_Decimal, boolean), Reference);

-- this procedure subtracts Right from Left, storing the
-- result in Result
-- the "error" boolean is set to true on overflow
procedure subtract (Result : in out Max_Decimal;
                  Left, Right : Max_Decimal;
                  error : in out boolean);
pragma interface (assembler, subtract);
pragma import_procedure (subtract, "SUB",
                        (Max_Decimal, Max_Decimal,
                         Max_Decimal, boolean), Reference);

-- this procedure multiplies Left by Right, and stores the
-- result in Result
-- the "error" boolean is set to true on overflow
procedure multiply (Result : in out Max_Decimal;
                  Left, Right : Max_Decimal;
                  error : in out boolean);
pragma interface (assembler, multiply);
pragma import_procedure (multiply, "MUL",
                        (Max_Decimal, Max_Decimal,
                         Max_Decimal, boolean), Reference);

-- this procedure divides Left by Right, storing the result
-- in Result
procedure divide (Result : in out Max_Decimal;
                 Left, Right : Max_Decimal;
                 Shift : in out integer;
                 error : in out boolean);
pragma interface (assembler, divide);
pragma import_procedure (divide, "DIV",
                        (Max_Decimal, Max_Decimal,
                         Max_Decimal, integer, boolean),
                        Reference);

```



```

-----
function max (Left, Right : decimal_digits) return
  decimal_digits is
begin
  if Left >= Right then
    return Left;
  else
    return Right;
  end if;
end max;

function To_SQL_Decimal_Not_Null (Value : SQL_Decimal_Not_Null2)
  return SQL_Decimal_Not_Null is
begin
  return (Value.scale, Value.Value);
end To_SQL_Decimal_Not_Null;

function To_SQL_Decimal (Value : SQL_Decimal_Not_Null2)
  return SQL_Decimal is
begin
  return (Value.scale, False, To_SQL_Decimal_Not_Null(Value));
end To_SQL_Decimal;

function To_SQL_Decimal_Not_Null2 (Value : SQL_Decimal_Not_Null)
  return SQL_Decimal_Not_Null2 is
begin
  return (Value.scale, Value.Value);
end To_SQL_Decimal_Not_Null2;

function To_SQL_Decimal_Not_Null2 (Value : SQL_Decimal)
  return SQL_Decimal_Not_Null2 is
begin
  if Value.Is_Null then
    raise null_value_error;
  else
    return To_SQL_Decimal_Not_Null2(Value.Value);
  end if;
end To_SQL_Decimal_Not_Null2;

function Null_SQL_Decimal return SQL_Decimal is
  Null_Holder : SQL_Decimal(0);
begin
  return Null_Holder;
end Null_SQL_Decimal;

function Shift (Value : SQL_Decimal_Not_Null;
  Scale : integer) return SQL_Decimal_Not_Null is
  Holder : SQL_Decimal_Not_Null := Value;
  error : boolean := false;
begin
  shft (Holder.Value, Value.Value, Scale, error);
  if error then
    raise Constraint_Error;
  end if;
  return Holder;
end Shift;

function Shift (Value : SQL_Decimal;
  Scale : integer) return SQL_Decimal is
begin
  if Value.Is_Null then
    return Null_SQL_Decimal;

```

```

        else
            return (Value.scale, False, Shift(Value.Value, Scale));
        end if;
    end Shift;

function Zero return SQL_Decimal_Not_Null is
begin
    return To_SQL_Decimal_Not_Null(0);
end Zero;

function Zero return SQL_Decimal is
begin
    return (0, False, Zero);
end Zero;

function One return SQL_Decimal_Not_Null is
begin
    return To_SQL_Decimal_Not_Null(1);
end One;

function One return SQL_Decimal is
begin
    return (0, False, One);
end One;

procedure Assign_With_Check (Left  : in out SQL_Decimal_Not_Null;
                             Right : SQL_Decimal_Not_Null;
                             Lower, Upper : SQL_Decimal_Not_Null2) is
    Holder : SQL_Decimal_Not_Null;
    error  : boolean := false;
begin
    if Right >= To_SQL_Decimal_Not_Null(Lower) and then
        Right <= To_SQL_Decimal_Not_Null(Upper) then
        if not (Left.scale = Right.scale) then
            shft(Holder.Value, Right.Value, (Left.scale - Right.scale),
                error);
            Left.Value := Holder.Value;
        else
            Left := Right;
        end if;
    else
        raise Constraint_Error;
    end if;
end Assign_With_Check;

procedure Assign_with_check
    (Left  : in out SQL_Decimal;
     Right : SQL_Decimal;
     Lower, Upper : SQL_Decimal_Not_Null2) is
    Holder : SQL_Decimal_Not_Null;
    error  : boolean := false;
begin
    if Right.Is_Null then
        Left.Is_Null := True;
    else
        if Right.Value >= To_SQL_Decimal_Not_Null(Lower) and then
            Right.Value <= To_SQL_Decimal_Not_Null(Upper) then
            Left.Is_Null := False;
            if not (Left.Value.scale = Right.Value.scale) then
                shft(Holder.Value, Right.Value.Value,
                    (Left.Value.scale - Right.Value.scale),
                    error);
                Left.Value.Value := Holder.Value;
            end if;
        else
            raise Constraint_Error;
        end if;
    end if;
end Assign_with_check;

```

```

        else
            Left.Value := Right.Value;
        end if;
    else
        raise Constraint_Error;
    end if;
end if;
end Assign_with_check;

function "=" (Left, Right : SQL_Decimal_Not_Null) return boolean is
    digs : integer;
    Holder : SQL_Decimal_Not_Null;
    error, result : boolean := false;
begin
    if Left.scale /= Right.scale then
        digs := abs(integer(Left.scale - Right.scale));
        if Left.scale > Right.scale then
            shft (Holder.Value, Right.Value, digs, error);
            if error then
                return False;
            end if;
            equal (Left.Value, Holder.Value, result);
        else
            shft (Holder.Value, Left.Value, digs, error);
            if error then
                return False;
            end if;
            equal (Holder.Value, Right.Value, result);
        end if;
    else
        equal (Left.Value, Right.Value, result);
    end if;
    return result;
end "=";

function "=" (Left, Right : SQL_Decimal)
    return boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return False;
    else
        if (Left.Value = Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end "=";

function Equals (Left, Right : SQL_Decimal)
    return Boolean_With_Unknown is
begin
    if Left.Is_Null or else Right.Is_Null then
        return Unknown;
    else
        if (Left.Value = Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end Equals;

```

```

function Not_Equals (Left, Right : SQL_Decimal)
  return Boolean_With_Unknown is
begin
  if Left.Is_Null or else Right.Is_Null then
    return Unknown;
  else
    if (Left.Value /= Right.Value) then
      return True;
    else
      return False;
    end if;
  end if;
end Not_Equals;

function "<" (Left, Right : SQL_Decimal_Not_Null) return boolean is
  digs : integer;
  Holder : SQL_Decimal_Not_Null;
  error, result : boolean := false;
begin
  if Left.scale /= Right.scale then
    digs := abs(integer(Left.scale - Right.scale));
    if Left.scale > Right.scale then
      shft (Holder.Value, Right.Value, digs, error);
      if error then
        if Right > Zero then
          return True;
        else
          return False;
        end if;
      end if;
      less_than (Left.Value, Holder.Value, result);
    else
      shft (Holder.Value, Left.Value, digs, error);
      if error then
        if Left < Zero then
          return True;
        else
          return False;
        end if;
      end if;
      less_than (Holder.Value, Right.Value, result);
    end if;
  else
    less_than (Left.Value, Right.Value, result);
  end if;
  return result;
end "<";

function "<" (Left, Right : SQL_Decimal)
  return boolean is
begin
  if Left.Is_Null or else Right.Is_Null then
    return False;
  else
    if (Left.Value < Right.Value) then
      return True;
    else
      return False;
    end if;
  end if;
end "<";

function "<" (Left, Right : SQL_Decimal)

```

```

    return Boolean_With_Unknown is
begin
    if Left.Is_Null or else Right.Is_Null then
        return Unknown;
    else
        if (Left.Value < Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end "<";

function ">" (Left, Right : SQL_Decimal_Not_Null) return boolean is
    digs : integer;
    Holder : SQL_Decimal_Not_Null;
    error, result : boolean := false;
begin
    if Left.scale /= Right.scale then
        digs := abs(integer(Left.scale - Right.scale));
        if Left.scale > Right.scale then
            shift (Holder.Value, Right.Value, digs, error);
            if error then
                if Right < Zero then
                    return True;
                else
                    return False;
                end if;
            end if;
            greater_than (Left.Value, Holder.Value, result);
        else
            shift (Holder.Value, Left.Value, digs, error);
            if error then
                if Left > Zero then
                    return True;
                else
                    return False;
                end if;
            end if;
            greater_than (Holder.Value, Right.Value, result);
        end if;
    else
        greater_than (Left.Value, Right.Value, result);
    end if;
    return result;
end ">";

function ">" (Left, Right : SQL_Decimal)
    return boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return False;
    else
        if (Left.Value > Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end ">";

function ">" (Left, Right : SQL_Decimal)
    return Boolean_With_Unknown is

```

```

begin
  if Left.Is_Null or else Right.Is_Null then
    return Unknown;
  else
    if (Left.Value > Right.Value) then
      return True;
    else
      return False;
    end if;
  end if;
end ">";

function "<=" (Left, Right : SQL_Decimal_Not_Null) return boolean is
  digs : integer;
  Holder : SQL_Decimal_Not_Null;
  error, result : boolean := false;
begin
  if Left.scale /= Right.scale then
    digs := abs(integer(Left.scale - Right.scale));
    if Left.scale > Right.scale then
      shft (Holder.Value, Right.Value, digs, error);
      if error then
        if Right > Zero then
          return True;
        else
          return False;
        end if;
      end if;
      less_than_equal (Left.Value, Holder.Value, result);
    else
      shft (Holder.Value, Left.Value, digs, error);
      if error then
        if Left < Zero then
          return True;
        else
          return False;
        end if;
      end if;
      less_than_equal (Holder.Value, Right.Value, result);
    end if;
  else
    less_than_equal (Left.Value, Right.Value, result);
  end if;
  return result;
end "<=";

function "<=" (Left, Right : SQL_Decimal)
  return boolean is
begin
  if Left.Is_Null or else Right.Is_Null then
    return False;
  else
    if (Left.Value <= Right.Value) then
      return True;
    else
      return False;
    end if;
  end if;
end "<=";

function "<=" (Left, Right : SQL_Decimal)
  return Boolean_With_Unknown is
begin

```

```

if Left.Is_Null or else Right.Is_Null then
    return Unknown;
else
    if (Left.Value <= Right.Value) then
        return True;
    else
        return False;
    end if;
end if;
end "<=";

function ">=" (Left, Right : SQL_Decimal_Not_Null) return boolean is
    digs : integer;
    Holder : SQL_Decimal_Not_Null;
    error, result : boolean := false;
begin
    if Left.scale /= Right.scale then
        digs := abs(integer(Left.scale - Right.scale));
        if Left.scale > Right.scale then
            shift (Holder.Value, Right.Value, digs, error);
            if error then
                if Right < Zero then
                    return True;
                else
                    return False;
                end if;
            end if;
            greater_than_equal (Left.Value, Holder.Value, result);
        else
            shift (Holder.Value, Left.Value, digs, error);
            if error then
                if Left > Zero then
                    return True;
                else
                    return False;
                end if;
            end if;
            greater_than_equal (Holder.Value, Right.Value, result);
        end if;
    else
        greater_than_equal (Left.Value, Right.Value, result);
    end if;
    return result;
end ">=";

function ">=" (Left, Right : SQL_Decimal)
    return boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return False;
    else
        if (Left.Value >= Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end ">=";

function ">=" (Left, Right : SQL_Decimal)
    return Boolean_With_Unknown is
begin
    if Left.Is_Null or else Right.Is_Null then

```

```

        return Unknown;
    else
        if (Left.Value >= Right.Value) then
            return True;
        else
            return False;
        end if;
    end if;
end ">=";

function Is_In_Base (Right : SQL_Decimal_Not_Null;
                    Lower, Upper : SQL_Decimal_Not_Null2)
    return boolean is
begin
    if Right >= To_SQL_Decimal_Not_Null(Lower) and then
        Right <= To_SQL_Decimal_Not_Null(Upper) then
        return True;
    else
        return False;
    end if;
end Is_In_Base;

function Is_In_Base (Right : SQL_Decimal;
                    Lower, Upper : SQL_Decimal_Not_Null2)
    return boolean is
begin
    if Right.Is_Null then
        return True;
    else
        if Right.Value >= To_SQL_Decimal_Not_Null(Lower) and then
            Right.Value <= To_SQL_Decimal_Not_Null(Upper) then
            return True;
        else
            return False;
        end if;
    end if;
end Is_In_Base;

function Is_Null(Value : SQL_Decimal) return boolean is
begin
    return Value.Is_Null;
end Is_Null;

function Not_Null(Value : SQL_Decimal) return boolean is
begin
    return not Value.Is_Null;
end Not_Null;

function "+" (Right : SQL_Decimal_Not_Null) return SQL_Decimal_Not_Null is
begin
    return Right;
end "+";

function "+" (Right : SQL_Decimal) return SQL_Decimal is
begin
    return Right;
end "+";

function "-" (Right : SQL_Decimal_Not_Null) return SQL_Decimal_Not_Null is
    Value : Max_Decimal;
begin
    inverse (Value, Right.Value);
    return (Right.Scale, Value);
end "-";

```



```

end "-";

function "-" (Right : SQL_Decimal) return SQL_Decimal is
begin
  if Right.Is_Null then
    return Null_SQL_Decimal;
  else
    return (Right.scale, False, -(Right.Value));
  end if;
end "-";

function "abs" (Right : SQL_Decimal_Not_Null) return SQL_Decimal_Not_Null is
Value : Max_Decimal;
begin
  absv (Value, Right.Value);
  return (Right.Scale, Value);
end "abs";

function "abs" (Right : SQL_Decimal) return SQL_Decimal is
begin
  if Right.Is_Null then
    return Null_SQL_Decimal;
  else
    return (Right.scale, False, abs(Right.Value));
  end if;
end "abs";

function "+" (Left, Right : SQL_Decimal_Not_Null)
return SQL_Decimal_Not_Null is
digs : integer;
Result, Holder : SQL_Decimal_Not_Null;
error : boolean := false;
begin
  if Left.scale /= Right.scale then
    digs := abs(integer(Left.scale - Right.scale));
    if Left.scale > Right.scale then
      Holder := Right;
      add (Result.Value, Left.Value, Shift(Holder, digs).Value,
          error);
    else
      Holder := Left;
      add (Result.Value, Shift(Holder, digs).Value, Right.Value,
          error);
    end if;
  else
    add (Result.Value, Left.Value, Right.Value, error);
  end if;
  if error then
    raise Constraint_Error;
  else
    return (max(Left.scale, Right.scale), Result.Value);
  end if;
end "+";

function "+" (Left, Right : SQL_Decimal)
return SQL_Decimal is
begin
  if Left.Is_Null or else Right.Is_Null then
    return Null_SQL_Decimal;
  else
    return (max(Left.scale, Right.scale), False,
            (Left.Value + Right.Value));
  end if;

```

```

end "+";

function "-" (Left, Right : SQL_Decimal_Not_Null)
  return SQL_Decimal_Not_Null is
    digs : integer;
    Result, Holder : SQL_Decimal_Not_Null;
    error : boolean := false;
begin
  if Left.scale /= Right.scale then
    digs := abs(integer(Left.scale - Right.scale));
    if Left.scale > Right.scale then
      Holder := Right;
      subtract (Result.Value, Left.Value, Shift(Holder, digs).Value,
        error);
    else
      Holder := Left;
      subtract (Result.Value, Shift(Holder, digs).Value,
        Right.Value, error);
    end if;
  else
    subtract (Result.Value, Left.Value, Right.Value, error);
  end if;
  if error then
    raise Constraint_Error;
  else
    return (max(Left.scale, Right.scale), Result.Value);
  end if;
end "-";

function "-" (Left, Right : SQL_Decimal)
  return SQL_Decimal is
begin
  if Left.Is_Null or else Right.Is_Null then
    return Null_SQL_Decimal;
  else
    return (max(Left.scale, Right.scale), False,
      (Left.Value - Right.Value));
  end if;
end "-";

function "*" (Left, Right : SQL_Decimal_Not_Null)
  return SQL_Decimal_Not_Null is
    Result : SQL_Decimal_Not_Null;
    error : boolean := false;
begin
  if (Left = Zero) then
    return Left;
  elsif (Right = Zero) then
    return Right;
  end if;
  if (Left.scale + Right.scale) > decimal_digits'last then
    raise Constraint_Error;
  end if;
  multiply (Result.Value, Left.Value, Right.Value, error);
  if error then
    raise Constraint_Error;
  end if;
  return ((Left.scale + Right.scale), Result.Value);
end "*";

function "*" (Left, Right : SQL_Decimal)
  return SQL_Decimal is
begin

```

```

if Left.Is_Null or else Right.Is_Null then
    return Null_SQL_Decimal;
else
    if Left.Value = Zero then
        return Left;
    elsif Right.Value = Zero then
        return Right;
    else
        return ((Left.scale + Right.scale), False,
            (Left.Value * Right.Value));
    end if;
end if;
end "*";

function "/" (Left, Right : SQL_Decimal_Not_Null)
    return SQL_Decimal_Not_Null is
    prec : decimal_digits := decimal_digits('last');
    Left_digs, Right_digs, Result_digs : integer;
    Right_Scale, Result_Scale : integer;
    Right_Holder, Result_Holder : SQL_Decimal_Not_Null;
    error : boolean := false;
begin
    if (Left = Zero) then
        return Left;
    end if;
    Right_Holder := Right;

    -- shift the BCD value in Right_Holder all the way to the
    -- right, eliminating trailing zeroes
    -- adjust the scale accordingly
    -- this will help to yield a result of maximum precision

    trailing_zeroes (Right_Holder.Value, prec, Right_digs);
    if Right_digs = decimal_digits('last') then
        raise Constraint_Error;
    else
        Right_digs := -Right_digs;
        Right_Holder := Shift (Right_Holder, Right_digs);
        Right_scale := Right.scale + Right_digs;
    end if;

    -- perform divide operation

    divide (Result_Holder.Value, Left.Value,
        Right_Holder.Value, Left_digs, error);

    if error then
        raise Constraint_Error;
    end if;

    -- if the scale of the result is outside the bounds of
    -- the available precision, shift the result left or
    -- right, accordingly

    Result_scale := Left.scale - Right_scale + Left_digs;
    if Result_scale > decimal_digits('last') then
        Result_digs := decimal_digits('last') - Result_scale;
        Result_scale := decimal_digits('last');
        Result_Holder := Shift (Result_Holder, Result_digs);
    elsif Result_Scale < 0 then
        Result_Holder := Shift (Result_Holder, abs(Result_Scale));
        Result_Scale := 0;
    end if;
end if;

```

```

        return (decimal_digits(Result_scale), Result_Holder.Value);
    end "/";

function "/" (Left, Right : SQL_Decimal)
    return SQL_Decimal is
begin
    if Left.Is_Null or else Right.Is_Null then
        return Null_SQL_Decimal;
    else
        return ("/"(Left.Value, Right.Value).scale, False,
            (Left.Value / Right.Value));
    end if;
end "/";

function "*" (Left : SQL_Decimal_Not_Null; Right : SQL_Int_Not_Null)
    return SQL_Decimal_Not_Null is
begin
    return (Left * To_SQL_Decimal_Not_Null(Right));
end "*";

function "*" (Left : SQL_Decimal; Right : SQL_Int_Not_Null)
    return SQL_Decimal is
begin
    if Left.Is_Null then
        return Null_SQL_Decimal;
    else
        return (Left.scale, False, (Left.Value * Right));
    end if;
end "*";

function "*" (Left : SQL_Decimal; Right : SQL_Int)
    return SQL_Decimal is
begin
    if Left.Is_Null or else Is_Null(Right) then
        return Null_SQL_Decimal;
    else
        return (Left.scale, False,
            (Left.Value * Without_Null_Base(Right)));
    end if;
end "*";

function "*" (Left : SQL_Int_Not_Null; Right : SQL_Decimal_Not_Null)
    return SQL_Decimal_Not_Null is
begin
    return (To_SQL_Decimal_Not_Null(Left) * Right);
end "*";

function "*" (Left : SQL_Int_Not_Null; Right : SQL_Decimal)
    return SQL_Decimal is
begin
    if Right.Is_Null then
        return Null_SQL_Decimal;
    else
        return (Right.scale, False, (Left * Right.Value));
    end if;
end "*";

function "*" (Left : SQL_Int; Right : SQL_Decimal)
    return SQL_Decimal is
begin
    if Right.Is_Null or else Is_Null(Left) then
        return Null_SQL_Decimal;
    else

```

```

        return (Right.scale, False,
                (Without_Null_Base(Left) * Right.Value));
    end if;
end "*";

function "/" (Left : SQL_Decimal_Not_Null; Right : SQL_Int_Not_Null)
    return SQL_Decimal_Not_Null is
begin
    return (Left / To_SQL_Decimal_Not_Null(Right));
end "/";

function "/" (Left : SQL_Decimal; Right : SQL_Int_Not_Null)
    return SQL_Decimal is
begin
    if Left.Is_Null then
        return Null_SQL_Decimal;
    else
        return ("/"(Left.Value, Right).scale, False,
                (Left.Value / Right));
    end if;
end "/";

function "/" (Left : SQL_Decimal; Right : SQL_Int)
    return SQL_Decimal is
begin
    if Left.Is_Null or else Is_Null(Right) then
        return Null_SQL_Decimal;
    else
        return ("/"(Left.Value, Without_Null_Base(Right)).scale,
                False, (Left.Value / Without_Null_Base(Right)));
    end if;
end "/";

function To_SQL_Decimal_Not_Null (Right : SQL_Int_Not_Null)
    return SQL_Decimal_Not_Null is
    Holder : SQL_Decimal_Not_Null;
begin
    integer_to_decimal(Holder.Value, BCD_Int_Type(Right));
    return Holder;
end To_SQL_Decimal_Not_Null;

function To_SQL_Decimal_Not_Null (Right : SQL_Double_Precision_Not_Null)
    return SQL_Decimal_Not_Null is
    Value : Max_Decimal;
    Scale : decimal_digits;
    prec : integer := SQL_Double_Precision_Not_Null'digits;
    exp : integer;
    temp_string : string(1..prec+6);
    Number_String : SQL_Char_Not_Null(1..decimal_digits'last+1) :=
        (1 => '+', 2..decimal_digits'last+1 => '0');
begin
    put(to => temp_string,
        item => Right,
        aft => prec - 1,
        exp => 3);
    exp := integer'value(temp_string(prec+4..prec+6));
    temp_string(3..prec+1) := temp_string(4..prec+2);
    if exp < prec-1 then
        if exp-prec+1 < -(decimal_digits'last) then
            raise Constraint_Error;
        else
            Scale := abs(exp - (prec - 1));
        end if;
    end if;
end To_SQL_Decimal_Not_Null;

```

```

        Number_String(decimal_digits'last+2-prec..
            decimal_digits'last+1) :=
        To_SQL_Char_Not_Null(temp_string(2..prec+1));
    end if;
else
    if exp > decimal_digits'last-1 then
        raise Constraint_Error;
    else
        Scale := 0;
        Number_String(decimal_digits'last+1-exp..
            decimal_digits'last-exp+prec) :=
        To_SQL_Char_Not_Null(temp_string(2..prec+1));
    end if;
end if;
if temp_string(1) = '-' then
    Number_String(1) := '-';
end if;
numeric_string_to_decimal (Value, Number_String);
return (Scale, Value);
end To_SQL_Decimal_Not_Null;

```

```

function To_SQL_Decimal_Not_Null (Right : SQL_Char_Not_Null)
return SQL_Decimal_Not_Null is
    temp : SQL_Char_Not_Null(1..decimal_digits'last+1);
    frst, lst, indx, lngth : integer;
    temp_scale : decimal_digits := 0;
    decimal_found : boolean := false;
    Value : Max_Decimal;
begin
    lst := Right'length;
    if Right(1) = '-' or else Right(1) = '+' then
        temp(1) := Right(1);
        frst := 2;
    elsif Right(1) = '.' then
        temp(1) := '+';
        frst := 2;
    else
        temp(1) := '+';
        frst := 1;
    end if;
    lngth := 1;
    for indx in frst..lst loop
        lngth := lngth + 1;
        if Right(indx) = '.' then
            if decimal_found then
                raise Constraint_Error;
            else
                decimal_found := true;
                temp_scale := decimal_digits(lst - indx);
                lngth := lngth - 1;
            end if;
        elsif ((Right(indx) = '0') or else
            (Right(indx) = '1') or else
            (Right(indx) = '2') or else
            (Right(indx) = '3') or else
            (Right(indx) = '4') or else
            (Right(indx) = '5') or else
            (Right(indx) = '6') or else
            (Right(indx) = '7') or else
            (Right(indx) = '8') or else
            (Right(indx) = '9')) then
            temp(lngth) := Right(indx);
        else

```

```

        raise Constraint_Error;
    end if;
end loop;
if length < decimal_digits'last+1 then
    temp := temp(1..1) & (2..decimal_digits'last+2-length => '0') &
        temp(2..length);
end if;
numeric_string_to_decimal (Value, temp);
return (temp_scale, Value);
end To_SQL_Decimal_Not_Null;

function To_SQL_Decimal (Right : SQL_Int_Not_Null) return SQL_Decimal is
begin
    return (0, False, To_SQL_Decimal_Not_Null(Right));
end To_SQL_Decimal;

function To_SQL_Decimal (Right : SQL_Int) return SQL_Decimal is
begin
    if Is_Null(Right) then
        return Null_SQL_Decimal;
    else
        return (0, False, To_SQL_Decimal_Not_Null(
            Without_Null_Base(Right)));
    end if;
end To_SQL_Decimal;

function To_SQL_Decimal (Right : SQL_Double_Precision_Not_Null)
    return SQL_Decimal is
begin
    return (To_SQL_Decimal_Not_Null(Right).scale, False,
        To_SQL_Decimal_Not_Null(Right));
end To_SQL_Decimal;

function To_SQL_Decimal (Right : SQL_Double_Precision) return SQL_Decimal is
begin
    if Is_Null(Right) then
        return Null_SQL_Decimal;
    else
        return (To_SQL_Decimal_Not_Null(Without_Null_Base(Right)).scale,
            False, To_SQL_Decimal_Not_Null(Without_Null_Base(Right)));
    end if;
end To_SQL_Decimal;

function To_SQL_Decimal (Right : SQL_Char_Not_Null)
    return SQL_Decimal is
begin
    return (To_SQL_Decimal_Not_Null(Right).scale, False,
        To_SQL_Decimal_Not_Null(Right));
end To_SQL_Decimal;

function To_SQL_Decimal (Right : SQL_Char)
    return SQL_Decimal is
begin
    if Is_Null(Right) then
        return Null_SQL_Decimal;
    else
        return (To_SQL_Decimal_Not_Null(Without_Null_Base(Right)).scale,
            False, To_SQL_Decimal_Not_Null(Without_Null_Base(Right)));
    end if;
end To_SQL_Decimal;

procedure Assign_To_SQL_Decimal (bound : in out SQL_Decimal_Not_Null2;
    sign : Sign_Character;

```

```

                                integral, scale : Numeric_String;
                                in_scale : decimal_digits) is
subtype new_char is SQL_Char_Not_Null(1..integral'length+scale'length);
Length : integer := integral'length + scale'length;
Number_String : SQL_Char_Not_Null(1..Length+2);
function unc is new unchecked_conversion (source => Numeric_String,
                                           target => new_char);

begin
  if Length > decimal_digits'last then
    raise Constraint_Error;
  and if;
  Number_String := unc(integral & scale) & "00";
  if sign = '-' then
    Number_String(1..Length+2) := "-" &
                                Number_String(1..Length-in_scale) &
                                "." &
                                Number_String(Length-in_scale+1..Length);
  else
    Number_String(1..Length+2) := "+" &
                                Number_String(1..Length-in_scale) &
                                "." &
                                Number_String(Length-in_scale+1..Length);
  and if;
  bound := To_SQL_Decimal_Not_Null2(
           To_SQL_Decimal_Not_Null(Number_String));
end Assign_To_SQL_Decimal;

function To_SQL_Int_Not_Null (Right : SQL_Decimal_Not_Null)
return SQL_Int_Not_Null is
Holder : BCD_Int_Type;
Decimal_Holder : SQL_Decimal_Not_Null;
error : boolean := false;
begin
  if Right.scale > 0 then
    Decimal_Holder := Right;
    decimal_to_integer(Holder, Shift(Decimal_Holder,
                                     -integer(Right.Scale)).Value, error);
  else
    decimal_to_integer(Holder, Right.Value, error);
  and if;
  if error then
    raise Constraint_Error;
  else
    return SQL_Int_Not_Null(Holder);
  and if;
end To_SQL_Int_Not_Null;

function To_SQL_Int_Not_Null (Right : SQL_Decimal)
return SQL_Int_Not_Null is
begin
  if Right.Is_Null then
    raise Null_Value_Error;
  else
    return To_SQL_Int_Not_Null(Right.Value);
  and if;
end To_SQL_Int_Not_Null;

function To_SQL_Int (Right : SQL_Decimal)
return SQL_Int is
begin
  if Right.Is_Null then
    return Null_SQL_Int;
  else

```



```

        return With_Null_Base(To_SQL_Int_Not_Null(Right.Value));
    end if;
end To_SQL_Int;

function To_SQL_Double_Precision_Not_Null (Right : SQL_Decimal_Not_Null)
return SQL_Double_Precision_Not_Null is
    indx, lngth : integer;
    Number_String : SQL_Char_Not_Null(1..decimal_digits'last+1);
    temp_holder : integer;
    prec : integer := SQL_Double_Precision_Not_Null'digits;
begin
    decimal_to_numeric_string (Number_String, Right.Value);
    indx := 2;
    while ((indx < decimal_digits'last+2) and then
        (Number_String(indx) = '0')) loop
        indx := indx + 1;
    end loop;
    if indx = decimal_digits'last+2 then
        return 0.0;
    end if;
    if indx < decimal_digits'last+3-prec then
        temp_holder := integer'value(To_String(Number_String(
            indx..indx+prec-1)));
        lngth := prec-1;
    else
        temp_holder := integer'value(To_String(Number_String(
            indx..decimal_digits'last+1)));
        lngth := decimal_digits'last+1-indx;
    end if;
    if Number_String(1) = '-' then
        temp_holder := -temp_holder;
    end if;
    if Right.Scale = 0 then
        return (SQL_Double_Precision_Not_Null(temp_holder) * (10.0 ** (
            decimal_digits'last + 1 - indx - lngth)));
    else
        return (SQL_Double_Precision_Not_Null(temp_holder) * (10.0 **
            (decimal_digits'last + 1 - indx - lngth - integer(Right.scale))));
    end if;
end To_SQL_Double_Precision_Not_Null;

function To_SQL_Double_Precision_Not_Null (Right : SQL_Decimal)
return SQL_Double_Precision_Not_Null is
begin
    if Right.Is_Null then
        raise Null_Value_Error;
    else
        return To_SQL_Double_Precision_Not_Null(Right.Value);
    end if;
end To_SQL_Double_Precision_Not_Null;

function To_SQL_Double_Precision (Right : SQL_Decimal)
return SQL_Double_Precision is
begin
    if Right.Is_Null then
        return Null_SQL_Double_Precision;
    else
        return With_Null_Base(To_SQL_Double_Precision_Not_Null(
            Right.Value));
    end if;
end To_SQL_Double_Precision;

function To_String (Right : SQL_Decimal_Not_Null) return string is

```

```

    Holder : SQL_Char_Not_Null(1..decimal_digits'last+3);
    indx : integer;
begin
    decimal_to_numeric_string (Holder, Right.Value);
    if Holder(1) = '+' then
        Holder(1) := ' ';
    end if;
    if Right.scale > 0 then
        Holder(decimal_digits'last+3-Right.scale..
            decimal_digits'last+2) :=
            Holder(decimal_digits'last+2-Right.scale..
                decimal_digits'last+1);
        Holder(decimal_digits'last+2-Right.scale) := '.';
        Holder(3..decimal_digits'last+3) :=
            Holder(2..decimal_digits'last+2);
        Holder(2) := '0';
        indx := 2;
        while (Holder(indx) = '0') loop
            indx := indx + 1;
        end loop;
        if Holder(indx) = '.' then
            indx := indx - 1;
        end if;
        return To_String(Holder(1..1) &
            Holder(indx..decimal_digits'last+3));
    else
        indx := 2;
        while (Holder(indx) = '0' and then
            indx < decimal_digits'last+2) loop
            indx := indx + 1;
        end loop;
        if indx = decimal_digits'last+2 then
            return " 0";
        else
            return To_String(Holder(1..1) &
                Holder(indx..decimal_digits'last+1));
        end if;
    end if;
end To_String;

function To_String (Right : SQL_Decimal) return string is
begin
    if Right.Is_Null then
        raise Null_Value_Error;
    else
        return To_String(Right.Value);
    end if;
end To_String;

function To_SQL_Char_Not_Null (Right : SQL_Decimal_Not_Null)
return SQL_Char_Not_Null is
    Holder : SQL_Char_Not_Null(1..decimal_digits'last+3);
    indx : integer;
begin
    decimal_to_numeric_string (Holder, Right.Value);
    if Holder(1) = '+' then
        Holder(1) := ' ';
    end if;
    if Right.scale > 0 then
        Holder(decimal_digits'last+3-Right.scale..
            decimal_digits'last+2) :=
            Holder(decimal_digits'last+2-Right.scale..
                decimal_digits'last+1);
    end if;
end To_SQL_Char_Not_Null;

```

```

Holder(decimal_digits'last+2-Right.scale) := 0;
Holder(3..decimal_digits'last+3) :=
  Holder(2..decimal_digits'last+2);
Holder(2) := '0';
indx := 2;
while (Holder(indx) = '0') loop
  indx := indx + 1;
end loop;
if Holder(indx) = '0' then
  indx := indx - 1;
end if;
return Holder(1..1) & Holder(indx..decimal_digits'last+3);
else
  indx := 2;
while (Holder(indx) = '0' and then
  indx < decimal_digits'last+2) loop
  indx := indx + 1;
end loop;
if indx = decimal_digits'last+2 then
  return "0";
else
  return Holder(1..1) & Holder(indx..decimal_digits'last+1);
end if;
end if;
end To_SQL_Char_Not_Null;

function To_SQL_Char_Not_Null (Right : SQL_Decimal)
return SQL_Char_Not_Null is
begin
  if Right.Is_Null then
    raise Null_Value_Error;
  else
    return To_SQL_Char_Not_Null(Right.Value);
  end if;
end To_SQL_Char_Not_Null;

function To_SQL_Char (Right : SQL_Decimal)
return SQL_Char is
begin
  if Right.Is_Null then
    return Null_SQL_Char;
  else
    return With_Null_Base(To_SQL_Char_Not_Null(Right.Value));
  end if;
end To_SQL_Char;

function Width (Right : SQL_Decimal_Not_Null) return integer is
begin
  return To_SQL_Char_Not_Null(Right)'length;
end Width;

function Width (Right : SQL_Decimal) return integer is
begin
  if Right.Is_Null then
    raise Null_Value_Error;
  else
    return Width(Right.Value);
  end if;
end Width;

function Integral_Digits (Right : SQL_Decimal_Not_Null)
return decimal_digits is
begin

```

```

    return decimal_digits(decimal_digits'last-integer(Right.scale));
end Integral_Digits;

function Integral_Digits (Right : SQL_Decimal)
    return decimal_digits is
begin
    return Integral_Digits(Right.Value);
end Integral_Digits;

function Scale (Right : SQL_Decimal_Not_Null)
    return decimal_digits is
begin
    return Right.scale;
end Scale;

function Scale (Right : SQL_Decimal)
    return decimal_digits is
begin
    return Scale(Right.Value);
end Scale;

function Fore (Right : SQL_Decimal_Not_Null)
    return positive is
    integral, digs : integer;
begin
    integral := decimal_digits'last-integer(Right.Scale);
    leading_zeros (Right.Value, integral, digs);
    digs := integral - digs;
    if digs = 0 then
        return 1;
    else
        return positive(digs);
    end if;
end Fore;

function Fore (Right : SQL_Decimal) return positive is
begin
    if Right.Is_Null then
        raise Null_Value_Error;
    end if;
    return Fore(Right.Value);
end Fore;

function Aft (Right : SQL_Decimal_Not_Null) return positive is
    digs : integer;
begin
    if Right.Scale = 0 then
        return 1;
    else
        trailing_zeros (Right.Value, Right.Scale, digs);
        digs := integer(Right.Scale) - digs;
        if digs = 0 then
            return 1;
        else
            return positive(digs);
        end if;
    end if;
end Aft;

function Aft (Right : SQL_Decimal) return positive is
begin
    if Right.Is_Null then
        raise Null_Value_Error;

```

```

        end if;
        return Aft(Right.Value);
    end Aft;

    function Machine_Rounds (Right : SQL_Decimal_Not_Null)
        return boolean is
    begin
        return True;
    end Machine_Rounds;

    function Machine_Rounds (Right : SQL_Decimal)
        return boolean is
    begin
        return True;
    end Machine_Rounds;

    function Machine_Overflows (Right : SQL_Decimal_Not_Null)
        return boolean is
    begin
        return True;
    end Machine_Overflows;

    function Machine_Overflows (Right : SQL_Decimal)
        return boolean is
    begin
        return True;
    end Machine_Overflows;

package body SQL_Decimal_Ops is

    lower_bound : SQL_Decimal_Not_Null2(in_scale);
    upper_bound : SQL_Decimal_Not_Null2(in_scale);

    procedure Assign (Left : in out Without_Null_Type;
                     Right : Without_Null_Type) is
    begin
        Assign_with_check (Left, Right, lower_bound, upper_bound);
    end Assign;

    procedure Assign (Left : in out With_Null_Type;
                     Right : With_Null_Type) is
    begin
        Assign_with_check (Left, Right, lower_bound, upper_bound);
    end Assign;

    function Is_In (Right : Without_Null_Type)
        return boolean is
    begin
        return Is_In_Base(Right, lower_bound, upper_bound);
    end Is_In;

    function Is_In (Right : With_Null_Type)
        return boolean is
    begin
        return Is_In_Base(Right, lower_bound, upper_bound);
    end Is_In;

    function With_Null (Value : Without_Null_Type)
        return With_Null_Type is
    begin
        return To_SQL_Decimal(To_SQL_Decimal_Not_Null2(Value));
    end With_Null;

```

```

function Without_Null (Value : With_Null_Type)
    return Without_Null_Type is
begin
    return To_SQL_Decimal_Not_Null(To_SQL_Decimal_Not_Null2(Value));
end Without_Null;

begin

    Assign_To_SQL_Decimal(lower_bound, first_sign, first_integral,
                          first_fractional, in_scale);

    Assign_To_SQL_Decimal(upper_bound, last_sign, last_integral,
                          last_fractional, in_scale);

end SQL_Decimal_Ops;

end SQL_Decimal_Pkg;

```

C.20 SQL_Decimal Assembler Support (VAX)

```

; -----
;
; PROCEDURE I2D
;
; procedure integer_to_decimal (Value : in out Max_Decimal;
;                               Right : integer);
;
; -- this procedure converts an integer into a packed decimal
; -- number 31 digits long
;
; -----
;
; .PSECT I2D
; .ENTRY I2D ^M<R2, R3>
; CVTLP  @8(AP), #31, @4(AP)
; RET
;
; -----
;
; PROCEDURE D2I
;
; procedure decimal_to_integer (Value : in out integer;
;                               Right : Max_Decimal;
;                               error : in out boolean);
;
; -- this procedure converts a packed decimal number of 31
; -- digits into an integer
;
; -----
;
; .PSECT D2I
; .ENTRY D2I ^M<R2, R3>
; CVTLP  #31, @8(AP), @4(AP)
; BVS   D2IERR
; RET
D2IERR: MOVL  #1, @12(AP)
; RET
;
; -----
;
; PROCEDURE NS2D
;
; procedure numeric_string_to_decimal (Value : in out Max_Decimal;
;                                       Right : string);
;
;
;

```

```

; -- this procedure converts a numeric string of 31 digits and a
; -- sign from leading separate numeric format into a packed
; -- decimal number of 31 digits
;
;-----
.PSECT NS2D
.ENTRY NS2D ^M<R2, R3>
CVTSP #31,@8(AP),#31,@4(AP)
RET
;-----
;
; PROCEDURE D2NS
;
; procedure decimal_to_numeric_string (Value : in out string;
;                                     Right : Max_Decimal);
;
; -- this procedure converts a packed decimal number of 31 digits
; -- into a numeric string in leading separate numeric format
;
;-----
.PSECT D2NS
.ENTRY D2NS ^M<R2, R3>
CVTSP #31,@8(AP),#31,@4(AP)
RET
;-----
;
; PROCEDURE LZ
;
; procedure leading_zeroes (Value : Max_Decimal;
;                           integ : integer;
;                           digs : in out integer);
;
; -- this procedure returns the number of leading zeroes in the
; -- first "integ" digits of the packed decimal number
;
;-----
.PSECT LZ
.ENTRY LZ ^M<R2, R3, R4, R5, R6, R7, R8>
MOVL @8(AP),R4
MOVL 4(AP),R5
CLRRL R8
LOOP: INCL R8
MOVB (R5),R6
BICL3 #^XFFFFFF0F,R6,R7
CMPB #^X00,R7
BNEQ DONE
DECL R4
CMPB #^X00,R4
BEQL DONE3
INCL R8
BICL3 #^XFFFFFFF0,R6,R7
CMPB #^X00,R7
BNEQ DONE
DECL R4
CMPB #^X00,R4
BEQL DONE3
INCL R5
BRB LOOP
DONE: DECL R8
DONE3: MOVL R8,@12(AP)
RET
;-----
;

```

```

; PROCEDURE TZ
;
; procedure trailing_zeros (Value : Max_Decimal;
;                          scal  : decimal_digits;
;                          digs  : in out integer);
;
; -- this procedure returns the number of trailing zeroes in
; -- the last "scal" digits of the packed decimal number
;
; -----

```

```

.PSECT TZ
.ENTRY TZ ^M<R2, R3, R4, R5, R6, R7, R8>
MOVL  @8(AP), R4
MOVL  4(AP), R5
ADDL  #15, R5
MOVB  (R5), R6
CLRL  R8
LOOP1: INCL  R8
      BICL3 #^XFFFFFF0F, R6, R7
      CMPB #^X00, R7
      BNEQ DONE1
      DECL  R4
      CMPB #^X00, R4
      BEQL DONE2
      DECL  R5
      MOVB (R5), R6
      INCL  R8
      BICL3 #^XFFFFFFF0, R6, R7
      CMPB #^X00, R7
      BNEQ DONE1
      DECI  R4
      CMPB #^X00, R4
      BEQL DONE2
      BRB  LOOP1
DONE1: DECL  R8
DONE2: MOVL  R8, @12(AP)
      RET

```

```

; PROCEDURE INV
;
; procedure inverse (Value : in out Max_Decimal;
;                  Right : Max_Decimal);
;
; -- this procedure returns the inverse of Right in Value
;
; -----

```

```

.PSECT INV
.ENTRY INV ^M<R2, R3, R4>
MOVCS #16, @8(AP), @4(AP)
MOVL  4(AP), R3
ADDL  #15, R3
MOVB  (R3), R2
BICL3 #^XFFFFFFF0, R2, R4
CMPB  #^X0F, R4
BNEQ  CNTNU
BICL2 #^X00000002, R2
BRB  INVEND
CNTNU: BICL2 #^X0000000E, R4
      CMPB #1, R4
      BEQL POS
      BICL2 #^X0000000F, R2
      BISL2 #^X0000000D, R2

```



```

        BRB     INVEND
POS:    BICL2  #^X0000000F,R2
        BISL2  #^X0000000C,R2
INVEND: MOVB   R2, (R3)
        RET
; -----
;
; PROCEDURE ABSV
;
; procedure absv (Value : in out Max_Decimal;
;                Right : Max_Decimal);
;
; -- this procedure returns the absolute_value of Right in Value
;
; -----
        .PSECT ABSV
        .ENTRY ABSV ^M<R2, R3>
        MOVCS  #16,@8(AP),@4(AP)
        MOVL   4(AP),R3
        ADDL   #15,R3
        MOVB   (R3),R2
        BICL2  #^X0000000F,R2
        BISL2  #^X0000000C,R2
        MOVB   R2, (R3)
        RET
; -----
;
; PROCEDURE SHFT
;
; procedure shft (Result : out Max_Decimal;
;                Value  : Max_Decimal;
;                scale  : integer;
;                error  : in out boolean);
;
; -- this procedure shifts the 31 digits of Value by "scale"
; -- digits.  if "scale" is positive, the shift is left.
; -- if "scale" is negative, the shift is right.  If overflow
; -- occurs on a left shift, then the error boolean is set to
; -- true.  The right shift rounds the remaining digits.
;
; -----
        .PSECT SHFTDATA
SDATA:  .BLKB  16
        .PSECT SHFT
        .ENTRY SHFT ^M<R2, R3, R4, R5>
        MOVL   @12(AP),R4
        ASHP   R4,#31,@8(AP),#5,#31,@4(AP)
        BVS    OVFLW
        RET
OVFLW:  MOVL   #1,@16(AP)
        RET
; -----
;
; PROCEDURE EQ
;
; procedure equal (Left, Right : Max_Decimal;
;                result      : in out boolean);
;
; -- this procedure compares Left and Right, and returns a result
; -- of true if they are equal, or false if they are not equal
;
; -----
        .PSECT EQ

```

```

        .ENTRY EQ ^M<R2, R3>
        CMPP3 #31,@4(AP),@8(AP)
        BEQL  EQTRU
        RET
EQTRU:  MOVL  #1,@12(AP)
        RET
; -----
;
; PROCEDURE LT
;
; procedure less_than (Left, Right : Max_Decimal;
;                      result      : in out boolean);
;
; -- this procedure compares Left and Right.  if Left is < Right
; -- then result is set to true
;
; -----
        .PSECT LT
        .ENTRY LT ^M<R2, R3>
        CMPP3 #31,@4(AP),@8(AP)
        BLSS  LTTRU
        RET
LTTRU:  MOVL  #1,@12(AP)
        RET
; -----
;
; PROCEDURE GT
;
; procedure greater_than (Left, Right : Max_Decimal;
;                        result      : in out boolean);
;
; -- this procedure compares Left and Right.  if Left > Right
; -- result is set to true.
;
; -----
        .PSECT GT
        .ENTRY GT ^M<R2, R3>
        CMPP3 #31,@4(AP),@8(AP)
        BGTR  GTTRU
        RET
GTTRU:  MOVL  #1,@12(AP)
        RET
; -----
;
; PROCEDURE LEQ
;
; procedure less_than_equal (Left, Right : Max_Decimal;
;                            result      : in out boolean);
;
; -- this procedure compares Left and Right.  if Left <= Right
; -- then result is set to true.
;
; -----
        .PSECT LEQ
        .ENTRY LEQ ^M<R2, R3>
        CMPP3 #31,@4(AP),@8(AP)
        BLEQ  LEQTRU
        RET
LEQTRU: MOVL  #1,@12(AP)
        RET
; -----
;
; PROCEDURE GEQ

```

```

;
; procedure greater_than_equal (Left, Right : Max_Decimal;
;                               result      : in out boolean);
;
; -- this procedure compares Left and Right.  if Left >= Right
; -- then result is set to true.
;
; -----
; .PSECT GEQ
; .ENTRY GEQ ^M<R2, R3>
; CMPP3 #31,@4(AP),@8(AP)
; BGEQ  GEQTRU
; RET
GEQTRU: MOVL #1,@12(AP)
; RET
; -----
;
; PROCEDURE ADD
;
; procedure add (Result : in out Max_Decimal;
;               Left, Right : Max_Decimal;
;               error  : in out boolean);
;
; -- this procedure adds Left and Right, and stores the result
; -- in Result.  if an overflow occurs during the operation, then
; -- "error" is set to true.
;
; -----
; .PSECT ADD
; .ENTRY ADD ^M<R2, R3, R4, R5>
; ADDP6 #31,@12(AP),#31,@8(AP),#31,@4(AP)
; BVS  ADDERR
; RET
ADDERR: MOVL #1,@16(AP)
; RET
; -----
;
; PROCEDURE SUB
;
; procedure subtract (Result : in out Max_Decimal;
;                   Left, Right : Max_Decimal;
;                   error  : in out boolean);
;
; -- this procedure subtracts Right from Left, and stores the result
; -- in Result.  if an overflow occurs during the operation, the
; -- "error" boolean is set to true.
;
; -----
; .PSECT SUB
; .ENTRY SUB ^M<R2, R3, R4, R5>
; SUBP6 #31,@12(AP),#31,@8(AP),#31,@4(AP)
; BVS  SUBERR
; RET
SUBERR: MOVL #1,@16(AP)
; RET
; -----
;
; PROCEDURE MUL
;
; procedure multiply (Result : in out Max_Decimal;
;                  Left, Right : Max_Decimal;
;                  error  : in out boolean);
;

```

```

; -- this procedure multiplies Left by Right, and stores the result
; -- in Result.  if an overflow occurs during the operation, the
; -- "error" boolean is set to true.
;
; -----
        .PSECT MUL
        .ENTRY MUL ^M<R2, R3, R4, R5>
        MULP    #31,@12(AP),#31,@8(AP),#31,@4(AP)
        BVS     MULERR
        RET
MULERR: MOVL   #1,@16(AP)
        RET
; -----
;
; PROCEDURE DIV
;
; procedure divide (Result : in out Max_Decimal;
;                  Left, Right : Max_Decimal;
;                  Shift : in out integer;
;                  error : in out boolean);
;
; -- this procedure divides Left by Right, and stores the result
; -- in Result.  no overflow can occur using this instruction.
; -- this procedure does not protect the application from the
; -- divide-by-zero run-time exception.
;
; -----
        .PSECT DIV
SHFTMP: .BLKB 16
        .ENTRY DIV ^M<R2, R3, R4, R5, R6, R7, R8>
        MOVL   #31,R4
        MOVL   8(AP),R5
        CLRL   R8
LOOPA:  INCL   R8
        MOVEB (R5),R6
        BICL3 #^XFFFFFF0F,R6,R7
        CMPB  #^X00,R7
        BNEQ  DONEA
        DECL  R4
        CMPB  #^X00,R4
        BEQL  DONEA
        INCL  R8
        BICL3 #^XFFFFFFF0,R6,R7
        CMPB  #^X00,R7
        BNEQ  DONEA
        DECL  R4
        CMPB  #^X00,R4
        BEQL  DONEA
        INCL  R5
        BRB   LOOPA
DONEA:  DECL  R8
        ASHP  R8,#31,@8(AP),#5,#31,SHFTMP
        DIVP  #31,@12(AP),#31,SHFTMP,#31,@4(AP)
        MOVL  R8,@16(AP)
        RET
        .END

```

C.21 SQL_Decimal Assembler Support (IBM)

Note: At the time this document was published, this code had not yet been fully tested. Electronically distributed versions of this code will be updated to reflect any changes made during testing.

```
ADASUP  CSECT
* -----
*
* PROCEDURE MI
*
* procedure mask_interrupts;
*
* -- this procedure turns off bit 37 in the PSW, to prevent
* -- the decimal overflow exception from causing an interrupt
*
* -----
MI      ENTRY  MI
        SAVE   (2,3)
        BALR   3,0
        USING  *,3
        SR     2,2          CLEAR R2
        O      2,=X'0B000000'  OR IN THE PROGRAM MASK
        SPM    2           TURN OFF BIT 37 OF THE PSW
        RETURN (2,3)
* -----
*
* PROCEDURE I2D
*
* procedure integer_to_decimal (Value : in out Max_Decimal;
*                               Right : integer);
*
* -- this procedure converts an integer into a packed decimal
* -- number 31 digits long
*
* -----
I2D     ENTRY  I2D
        SAVE   (2,5)
        BALR   5,0
        USING  *,5
        LM     2,3,0(1)     ADDRESS OF VALUE IN R2; RIGHT IN R3
        XC     0(8,2),0(2)  CLEAR UPPER 2 WORDS OF DEC RESULT
        CVD    3,8(2)      CONVERT INTEGER, STORE IN WRDS 3 & 4
        RETURN (2,5)
* -----
*
* PROCEDURE D2I
*
* procedure decimal_to_integer (Value : in out integer;
*                               Right : Max_Decimal;
*                               error : in out boolean);
*
* -- this procedure converts a packed decimal number of 31
* -- digits into an integer
*
* This procedure will cause a numeric error to occur in the
* application if the number to be converted falls outside the
* range -2147483648..2147483647
*
* -----
        ENTRY  D2I
```

```

D2I    SAVE    (2,5)
      BALR    5,0
      USING   *,5
      L      3,4(1)          ADDRESS OF RIGHT IN R3
      CP     0(16,3),LOWER(16) COMPARE INPUT TO MAX NEG INTEGER
      BL     D2IERR          IF LESS THAN, OVERFLOW WILL OCCUR
      CP     0(16,3),UPPER(16) COMPARE INPUT TO MAX POS INTEGER
      BH     D2IERR          IF GREATER THAN, OVERFLOW WILL OCCUR
      CVB    4,8(3)          CNVT LOWER 8 BYTES OF DECIMAL NUM
      ST     4,0(1)          STORE RESULT
      B      D2IRET          GO TO D2IRET
D2IERR L      2,=F'1'        SET VALUE OF ERROR BOOLEAN
      STC    2,8(1)          TO 'TRUE'
D2IRET RETURN (2,5)

```

```

* -----
*
* PROCEDURE NS2D
*
* procedure numeric_string_to_decimal (Value : in out Max_Decimal;
*                                     Right : string);
*
* -- this procedure converts a numeric string of 31 digits and a
* -- sign from leading separate numeric format into a packed
* -- decimal number of 31 digits
*
* -----

```

```

      ENTRY   NS2D
NS2D   SAVE    (2,5)
      BALR    5,0
      USING   *,5
      LM     2,3,0(1)        GET ADDRESSES OF PARMS
      PACK   0(9,2),1(16,3)  CK FRST 16 DIGS INTO FRST 9 BYTES
      SRP    0(9,2),1,5      SHFT LFT, SO 16 VALID DIGS IN 8 BYTS
      PACK   8(8,2),17(15,3) PACK LAST 15 DIGS INTO LAST 8 BYTES
      CLC    0(1,3),=X'4E'   CHECK SIGN
      BE     NS2DPOS         BRANCH TO MAKE RESULT POSITIVE
      NI     15(2),X'F0'     CLEAR SIGN DIGIT
      OI     15(2),X'0D'     MAKE RESULT NEGATIVE
      B      NS2DRET         RETURN AFTER MAKING RESULT NEGATIVE
NS2DPOS NI    15(2),X'F0'   CLEAR SIGN DIGIT
      OI     15(2),X'0C'     MAKE RESULT POSITIVE
NS2DRET RETURN (2,5)

```

```

* -----
*
* PROCEDURE D2NS
*
* procedure decimal_to_numeric_string (Value : in out string;
*                                     Right : Max_Decimal);
*
* -- this procedure converts a packed decimal number of 31 digits
* -- into a numeric string in leading separate numeric format
*
* -----

```

```

      ENTRY   D2NS
D2NS   SAVE    (2,5)
      BALR    5,0
      USING   *,5
      LM     2,3,0(1)        GET ADDRESSES OF PARMS
      UNPK   1(15,2),0(8,3)  UNPACK FIRST 14 DIGITS
      UNPK   15(15,2),7(8,3) UNPACK NEXT 14 DIGITS
      UNPK   29(3,2),14(2,3) UNPACK LAST 3 DIGITS
      SR     4,4             CLEAR R4
      IC     4,15(3)        GET SIGN OF INPUT

```

```

      N      4,=X'0000000F'      AND OUT NUMERIC PORTION OF BYTE
      CL      4,=X'0000000D'      CHECK THE SIGN
      BE      D2NSNEG             IF NEGATIVE, GO TO D2NSNEG
      MVI     0(2),X'4E'         MAKE POSITIVE
      B       D2NSTR             GO TO D2NSTR
D2NSNEG MVI     0(2),X'60'         MAKE NEGATIVE
D2NSTR  OI      31(2),X'F0'       MAKE LAST BYTE EBCDIC
      RETURN (2,5)

```

```

* -----
*
* PROCEDURE LZ
*
* procedure leading_zeros (Value : Max_Decimal;
*                          integ : integer;
*                          digs  : in out integer);
*
* -- this procedure returns the number of leading zeroes in the
* -- first "integ" digits of the packed decimal number
*
* -----

```

```

      ENTRY  LZ
LZ      SAVE   (2,8)
      BALR   8,0
      USING  *,8
      LM     2,3,0(1)           GET PARMS IN R2 AND R3
      BCTR   2,0                OFFSET ADDRESS BY ONE FOR LOOP
      SR     5,5                CLEAR R5
      SR     6,6                CLEAR R6
LOOP    LA    2,1(2)           GET NEXT BYTE TO LOOK AT
      LA    5,1(5)           ADD 1 TO R5 (COUNT OF ZERO DIGITS+1)
      IC    6,0(2)           GET ANOTHER BYTE OF PARM1
      SR     7,7                CLEAR R7
      SRDL  6,4                UPPER NIBBLE OF BYT IN R6, LWR IN R7
      C     6,ZERO            IF R6 IS ZERO, CONTINUE
      BNE   DONE              IF NOT, DONE
      BCTR  3,CONT            GET NEXT NIBBLE IF MORE TO SCAN
      B     DONE2             NO MORE TO SCAN
CONT    LA    5,1(5)           ADD 1 TO R5 (COUNT OF ZERO DIGITS+1)
      C     7,ZERO            IF R7 IS ZERO, CONTINUE
      BNE   DONE              IF NOT, DONE
      BCTR  3,LOOP            GOTO LOOP IF NOT FINISHED
      B     DONE2             NO NEED TO SUBT 1, ALL ZEROES
DONE    BCTR  5,0                R5 NOW CONTAINS COUNT OF ZERO DIGITS
DONE2   ST    5,8(1)           STORE RESULT
      RETURN (2,8)

```

```

* -----
*
* PROCEDURE TZ
*
* procedure trailing_zeros (Value : Max_Decimal;
*                            scal  : decimal_digits;
*                            digs  : in out integer);
*
* -- this procedure returns the number of trailing zeroes in
* -- the last "scal" digits of the packed decimal number
*
* -----

```

```

      ENTRY  TZ
TZ      SAVE   (2,8)
      BALR   8,0
      USING  *,8
      LM     2,3,0(1)           PARMS IN R2 AND R3
      LA    2,15(2)           GET ADDRESS OF LAST BYTE OF DEC NUMB

```

```

        IC      6,0(2)      GET LAST BYTE OF DEC NUMBER
        SRL     6,4        GET LAST DIGIT OF DEC NUMBER
        SR      5,5        CLEAR R5
LOOP1   LA      5,1(5)     ADD 1 TO R5 (COUNT OF ZERO DIGITS+1)
        C      6,ZERO     IF R6 IS ZERO, CONTINUE
        BNE    DONE1     IF NOT, DONE
        BCT   3,CONT1    GET NEXT BYTE IF MORE TO SCAN
        B     DONE3     NO MORE TO SCAN
CONT1  BCTR    2,0        GET ADDRESS OF NEXT BYTE OF DEC NUMB
        IC     6,0(2)     GET PREV BYTE OF DEC DIGIT
        SR     7,7        CLEAR R7 FOR SHIFT
        SRDL   6,4        UPPER NIBBLE => R6, LOWER => R7
        LA     5,1(5)     ADD 1 TO R5 (COUNT OF ZERO DIGITS+1)
        C      7,ZERO     IF R7 IS ZERO, CONTINUE
        BNE    DONE1     IF NOT, DONE
        BCT   3,LOOP1    GO TO LOOP1 IF MORE TO SCAN
        B     DONE3     NO NEED TO SUBT 1, ALL ZEROES
DONE1  BCTR    5,0        R5 NOW CONTAINS COUNT OF ZERO DIGITS
DONE3  ST      5,8(1)    STORE RESULT
        RETURN (2,8)

```

```

* -----
*
* PROCEDURE INV
*
* procedure inverse (Value : in out Max_Decimal;
*                   Right : Max_Decimal);
*
* -- this procedure returns the inverse of Right in Value
*
* -----

```

```

INV     ENTRY   INV
        SAVE   (2,6)
        BALR  6,0
        USING *,6
        LM    2,3,0(1)   GET ADDRESSES OF PARAMS
        MVC   0(16,2),0(3) MOVE INPUT TO OUTPUT
        IC    4,15(2)    LOAD LAST BYTE OF DEC NUMBER
        SR    5,5        CLEAR R5 FOR SHIFT
        SRDL  4,4        SHIFT RIGHT SO ONLY SIGN IN R5
        C     5,POSZCON  IS SIGN AN 'F'
        BNE   CNTNU     GO TO CNTNU IF NOT
        L     5,NEGCON   ELSE MAKE THE SIGN NEGATIVE
        B     INVEND    GO TO END
CNTNU   SLL   5,3        SHIFT TO SEE LOW ORDER BIT OF SIGN
        C     5,ZERO     IF LOW ORDER BIT IS ZERO, NUM IS POS
        BNE   POS       IF LOW ORDER BIT IS ONE, NUM IS NEG
        L     5,NEGCON   DEC NUM IS POS => MAKE NEG
        B     INVEND    GO TO END
POS     L     5,POSCON   DEC NUM IS NEG => MAKE POS
INVEND  SLDL  4,4        SHIFT LEFT SO LOW ORDER BYTE IN R4
        STC   4,15(2)   STORE LOW ORDER BYTE INTO DEC NUM
INVRET  RETURN (2,6)

```

```

* -----
*
* PROCEDURE ABSV
*
* procedure absv (Value : in out Max_Decimal;
*                Right : Max_Decimal);
*
* -- this procedure returns the absolute_value of Right in Value
*
* -----

```

```

        ENTRY   ABSV

```



```

ABSV  SAVE  (2,4)
      BALR  4,0
      USING *,4
      LM    2,3,0(1)          GET ADDRESSES OF PARAMS
      MVC   0(16,2),0(3)     MOVE INPUT TO OUTPUT
      NI    15(2),X'F0'      CLEAR SIGN
      OI    15(2),X'0C'      MAKE SIGN POS
      RETURN (2,4)

```

```

* -----
*
* PROCEDURE SHFT
*
* procedure shft (Result : out Max Decimal;
*                Value   : Max Decimal;
*                scale   : integer;
*                error   : in out boolean);
*
* -- this procedure shifts the 31 digits of Value by "scale"
* -- digits.  if "scale" is positive, the shift is left.
* -- if "scale" is negative, the shift is right.  If overflow
* -- occurs on a left shift, then the error boolean is set to
* -- true.  The right shift rounds the remaining digits.
*
* This subroutine expects that the Decimal Overflow mask in the PSW
* has been cleared to prevent the interrupt (bit pos 37).
*
* -----

```

```

      ENTRY  SHFT
SHFT  SAVE  (2,6)
      BALR  6,0
      USING *,6
      LM    2,4,0(1)          GET PARMS IN R2 THROUGH R4
      MVC   0(16,2),0(3)     MOVE THE INPUT TO THE OUTPUT
      L     3,=X'0F5'        LOAD LENGTH1 AND LENGTH2 FOR EX INST
      C     4,=F'64'         IF SHIFT COUNT > 64
      BH    SHFTERR          THEN COUNT OUTSIDE SHIFT RANGE
      C     4,=F'-64'        IF SHIFT COUNT < -64
      BL    SHFTERR          THEN COUNT OUTSIDE SHIFT RANGE
      C     4,=F'0'          IF SHIFT COUNT >= 0
      BNL   SHFTCNT          THEN CONTINUE, ELSE
      L     5,=F'64'         SHIFT IS TO RIGHT, 2ND OPND IS
      SR    5,4              64 - COUNT
      LR    4,5              GET COUNT IN R4
      SHFTCNT N 4,=X'00000FFF' ONLY LOWER 12 BITS CONTAINS COUNT
      STH   4,INST+4         STORE COUNT INTO SHIFT INSTRUCTION
      EX    3,INST           EXECUTE INSTRUCTION
      BO    SHFTERR          IF OVERFLOW, GO TO SHFTERR
      B     SHFTRET          GO TO SHFTRET
      SHFTERR LA 4,1         LOAD 'TRUE' IN R4
      STC   4,12(1)         STORE 'TRUE' INTO ERROR BOOLEAN
      SHFTRET RETURN (2,6)

```

```

* -----
*
* PROCEDURE EQ
*
* procedure equal (Left, Right : Max Decimal;
*                result      : in out boolean);
*
* -- this procedure compares Left and Right, and returns a result
* -- of true if they are equal, or false if they are not equal
*
* -----

```

```

      ENTRY  EQ

```

```

EQ      SAVE   (2,5)
        BALR   5,0
        USING  *,5
        LM     2,3,0(1)          GET ADDRESSES OF PARMs
        CP     0(16,2),0(16,3)   COMPARE TWO PACKED NUMS
        BNE    EQRET             RETURN 'FALSE' IF NOT EQ
        LA     2,1               LOAD 'TRUE' INTO R2
        STC    2,8(1)           STORE 'TRUE' INTO RESULT BOOLEAN
EQRET   RETURN (2,5)

```

```

* -----
*
* PROCEDURE LT
*

```

```

* procedure less_than (Left, Right : Max_Decimal;
*                      result      : in out boolean);
*
* -- this procedure compares Left and Right.  if Left is < Right
* -- then result is set to true
*
* -----

```

```

        ENTRY  LT
LT      SAVE   (2,5)
        BALR   5,0
        USING  *,5
        LM     2,3,0(1)          GET ADDRESSES OF PARMs
        CP     0(16,2),0(16,3)   COMPARE TWO PACKED NUMS
        BNL    LTRET            RETURN 'FALSE' IF NOT LT
        LA     2,1               LOAD 'TRUE' INTO R2
        STC    2,8(1)           STORE 'TRUE' INTO RESULT BOOLEAN
LTRET   RETURN (2,5)

```

```

* -----
*
* PROCEDURE GT
*

```

```

* procedure greater_than (Left, Right : Max_Decimal;
*                          result     : in out boolean);
*
* -- this procedure compares Left and Right.  if Left > Right
* -- result is set to true.
*
* -----

```

```

        ENTRY  GT
GT      SAVE   (2,5)
        BALR   5,0
        USING  *,5
        LM     2,3,0(1)          GET ADDRESSES OF PARMs
        CP     0(16,2),0(16,3)   COMPARE TWO PACKED NUMS
        BNH    GTRET            RETURN 'FALSE' IF NOT GT
        LA     2,1               LOAD 'TRUE' INTO R2
        STC    2,8(1)           STORE 'TRUE' INTO RESULT BOOLEAN
GTRET   RETURN (2,5)

```

```

* -----
*
* PROCEDURE LEQ
*

```

```

* procedure less_than_equal (Left, Right : Max_Decimal;
*                             result     : in out boolean);
*
* -- this procedure compares Left and Right.  if Left <= Right
* -- then result is set to true.
*
* -----

```

```

        ENTRY  LEQ

```

```

LEQ      SAVE    (2,5)
        BALR    5,0
        USING   *,5
        LM      2,3,0(1)          GET ADDRESSES OF PARMS
        CP      0(16,2),0(16,3)  COMPARE TWO PACKED NUMS
        BH      LEQRET           RETURN 'FALSE' IF NOT LEQ
        LA      2,1              LOAD 'TRUE' INTO R2
        STC     2,8(1)           STORE 'TRUE' INTO RESULT BOOLEAN
LEQRET   RETURN  (2,5)

```

```

* -----
*
* PROCEDURE GEQ
*
* procedure greater_than_equal (Left, Right : Max_Decimal;
*                               result      : in out boolean);
*
* -- this procedure compares Left and Right.  if Left >= Right
* -- then result is set to true.
*
* -----

```

```

        ENTRY   GEQ
GEQ      SAVE    (2,5)
        BALR    5,0
        USING   *,5
        LM      2,3,0(1)          GET ADDRESSES OF PARMS
        CP      0(16,2),0(16,3)  COMPARE TWO PACKED NUMS
        BL      GEQRET           RETURN 'FALSE' IF NOT GEQ
        LA      2,1              LOAD 'TRUE' INTO R2
        STC     2,8(1)           STORE 'TRUE' INTO RESULT BOOLEAN
GEQRET   RETURN  (2,5)

```

```

* -----
*
* PROCEDURE ADD
*
* procedure add (Result : in out Max_Decimal;
*               Left, Right : Max_Decimal;
*               error   : in out boolean);
*
* -- this procedure adds Left and Right, and stores the result
* -- in Result.  if an overflow occurs during the operation, then
* -- "error" is set to true.
*
* This subroutine expects that the Decimal Overflow mask in the PSW
* has been cleared to prevent the interrupt (bit pos 37).
*
* -----

```

```

        ENTRY   ADD
ADD      SAVE    (2,5)
        BALR    5,0
        USING   *,5
        LM      2,4,0(1)          GET ADDRESSES OF PARMS
        MVC     0(16,2),0(3)      MOVE 'LEFT' TO 'RESULT'
        AP      0(16,2),0(16,4)  ADD 'LEFT' AND 'RIGHT' IN PLACE
        BO      ADDRERR          GO TO ADDRERR ON OVERFLOW
        B       ADDRRET          GO TO ADDRRET
ADDRERR  LA      3,1              LOAD 'TRUE' INTO R3
        STC     3,12(1)          STORE 'TRUE' INTO ERROR BOOLEAN
ADDRRET  RETURN  (2,5)

```

```

* -----
*
* PROCEDURE SUB
*
* procedure subtract (Result : in out Max_Decimal;

```

```

*           Left, Right : Max_Decimal;
*           error      : in out boolean);
*
* -- this procedure subtracts Right from Left, and stores the result
* -- in Result.  if an overflow occurs during the operation, the
* -- "error" boolean is set to true.
*
* This subroutine expects that the Decimal Overflow mask in the PSW
* has been cleared to prevent the interrupt (bit pos 37).
*
* -----

```

```

      ENTRY  SUB
SUB   SAVE   (2,5)
      BALR   5,0
      USING  *,5
      LM     2,4,0(1)           GET ADDRESSES OF PARMS
      MVC    0(16,2),0(3)       MOVE 'LEFT' TO 'RESULT'
      SP     0(16,2),0(16,4)    SUBTRACT 'RIGHT' FROM 'LEFT'
      BO     SUBERR             GO TO SUBERR ON OVERFLOW
      B      SUBRET            GO TO SUBRET
SUBERR LA    3,1               LOAD 'TRUE' VALUE INTO R3
      STC    3,12(1)           STORE 'TRUE' INTO ERROR BOOLEAN
SUBRET RETURN (2,5)
* -----

```

```

* PROCEDURE MUL
*

```

```

* procedure multiply (Result : in out Max_Decimal;
*                   Left, Right : Max_Decimal;
*                   error      : in out boolean);
*

```

```

* -- this procedure multiplies Left by Right, and stores the result
* -- in Result.  if an overflow occurs during the operation, the
* -- "error" boolean is set to true.
*
* This procedure will cause a numeric error to occur in the application
* if there are not enough leading zeros in the multiplicand to
* accomodate the MP instruction.
*
* -----

```

```

      ENTRY  MUL
MUL   SAVE   (2,10)
      BALR   10,0
      USING  *,10
      LM     2,4,0(1)           GET ADDRESSES OF PARMS
      BCTR   3,0                OFFSET 'LEFT' TO PREPARE FOR LOOPA
      LA     5,31               GET NUMBER OF DIGITS TO SCAN
      SR     6,6                CLEAR R6
      SR     8,8                CLEAR R8
LOOPA LA    3,1(3)             GET ADDRESS OF NEXT BYTE TO SCAN
      LA     6,1(6)             ADD 1 TO R6 (COUNT OF ZERO DIGITS+1)
      IC     8,0(3)             GET ANOTHER BYTE OF LEFT
      SR     9,9                CLEAR R9
      SRDL   8,4                UPPER NIBBLE OF BYT IN R8, LWR IN R9
      C      8,ZERO             IF R8 IS ZERO, CONTINUE
      BNE   DONEA              IF NOT, DONE
      BCT   5,CONTA            CONTINUE IF MORE TO SCAN
      B     DONEA1             NO MORE TO SCAN
CONTA LA    6,1(6)             ADD 1 TO R6 (COUNT OF ZERO DIGITS+1)
      C      9,ZERO             IF R9 IS ZERO, CONTINUE
      BNE   DONEA              IF NOT, DONE
      BCT   5,LOOPA           GET NEXT BYTE IF MORE TO SCAN
      B     DONEA1             NO NEED TO SUBT 1, ALL ZEROES

```

DONEA	BCTR	6,0	R6 NOW CONTAINS COUNT OF ZERO DIGITS
DONEA1	LA	5,31	GET NUMBER OF DIGITS TO SCAN
	SR	7,7	CLEAR R7
	BCTR	4,0	OFFSET 'RIGHT' TO PREPARE FOR LOOPB
	SR	8,8	CLEAR R8
LOOPB	LA	4,1(4)	GET ADDRESS OF NEXT BYTE TO SCAN
	LA	7,1(7)	ADD 1 TO R7 (COUNT OF ZERO DIGITS+1)
	IC	8,0(4)	GET ANOTHER BYTE OF RIGHT
	SR	9,9	CLEAR R9
	SRDL	8,4	UPPER NIBBLE OF BYT IN R8, LWR IN R9
	C	8,ZERO	IF R8 IS ZERO, CONTINUE
	BNE	DONEB	IF NOT, DONE
	BCT	5,CONTB	SCAN NEXT NIBBLE IF MORE TO SCAN
	B	DONEB1	NO MORE TO SCAN
CONTB	LA	7,1(7)	ADD 1 TO R7 (COUNT OF ZERO DIGITS+1)
	C	9,ZERO	IF R9 IS ZERO, CONTINUE
	BNE	DONEB	IF NOT, DONE
	BCT	5,LOOPB	GET NEXT BYTE TO SCAN IF MORE
	B	DONEB1	NO NEED TO SUBT 1, ALL ZEROES
DONEB	BCTR	7,0	R7 NOW CONTAINS COUNT OF ZERO DIGITS
DONEB1	LM	3,4,4(1)	GET ADDRESSES OF LEFT AND RIGHT
	CR	6,7	WHICH OPERAND HAS MORE ZEROES?
	BH	MULV2	GO TO MULV2 IF RIGHT HAS MORE ZEROES
MULV1	SRL	6,1	CLEAR LOW ORDER BIT
	SLL	6,1	MAKE ODD # OF LEADING 0'S EVEN
	LR	8,6	LOAD R8 WITH # LEADING 0'S OF LEFT
	AR	8,7	ADD IN # LEADING 0'S OF RIGHT
	C	8,THYTTWO	IF NOT GREATER THAN 31, THEN
	BL	MULERR	MULTIPLY WILL RAISE AN EXCEPTION
	MVC	0(16,2),0(4)	LEFT HAS MORE ZEROES: MOVE RIGHT
	LA	8,32	TO RESULT
	SR	8,6	R8 CONTAINS NUM DIGITS IN LEFT
	SRL	8,1	DIVIDE NUM DIGS BY 2 TO GET NUM BYTS
	LA	8,1(8)	ADD IN REM TO GET NUM BYTES IN LEFT
	LA	3,16(3)	ADD 16 TO LEFT
	SR	3,8	SUB NUM BYTES TO GET CORRECT OFFSET
	BCTR	8,0	OFFSET LENGTH OF LEFT BY 1
	O	8,=X'000000F0'	OR IN LENGTH OF RESULT
	EX	8,MULV1A	EXECUTE MP INSTR USING LENGTHS IN R9
	B	MULRET	GO TO MULRET
MULV2	SRL	7,1	CLR LOW ORDER BIT, ODD # OF LDNG 0'S
	SLL	7,1	MAKE ODD # OF LEADING 0'S EVEN
	LR	8,7	LOAD R8 WITH # LEADING 0'S OF RIGHT
	AR	8,6	ADD IN # LEADING 0'S OF LEFT
	C	8,THYTTWO	IF NOT GREATER THAN 31, THEN
	BL	MULERR	MULTIPLY WILL RAISE AN EXCEPTION
	MVC	0(16,2),0(3)	RIGHT HAS MORE ZEROES: MOVE LEFT
	LA	8,32	TO RESULT
	SR	8,7	R8 CONTAINS NUM DIGITS IN RIGHT
	SRL	8,1	DIVIDE NUM DIGS BY 2 TO GET NUM BYTS
	LA	8,1(8)	ADD IN REM TO GET NUM BYTES IN RIGHT
	LA	4,16(4)	ADD 16 TO RIGHT
	SR	4,8	SUB NUM BYTES TO GET CORRECT OFFSET
	BCTR	8,0	OFFSET LENGTH OF RIGHT BY 1
	O	8,=X'000000F0'	OR IN LENGTH OF RESULT
	EX	8,MULV2A	EXECUTE MP INSTR USING LENGTHS IN R9
	B	MULRET	GO TO MULRET
MULERR	LA	3,1	PUT VALUE 'TRUE' INTO R3
	STC	3,12(1)	STORE R3 INTO ERROR
MULRET	RETURN	(2,10)	

* -----

*

* PROCEDURE DIV

```

*
* procedure divide (Result : in out Max_Decimal;
*                 Left, Right : Max_Decimal;
*                 Shift : in out integer;
*                 error : in out boolean);
*
* -- this procedure divides Left by Right, and stores the result
* -- in Result.  no overflow can occur using this instruction.
* -- this procedure does not protect the application from the
* -- divide-by-zero run-time exception.
*
* This procedure causes a numeric error exception to occur in
* the application if the result is too large for the space
* set aside for the quotient by the DP (divide packed) instruction,
* or if the actual number in the divisor is larger than 8 bytes.
*
*

```

```

-----
DIV      ENTRY  DIV
DIV      SAVE   (2,11)
        BALR   11,0
        USING  *,11
        LM    2,4,0(1)      GET ADDRESSES OF PARMS
        BCTR  3,0          OFFSET R3 TO PREPARE FOR LOOPC
        LA   10,31         GET NUMBER OF DIGITS TO SCAN
        SR   6,6          CLEAR R6
        SR   8,8          CLEAR R8
LOOPC    LA   3,1(3)       GET ADDRESS OF NEXT BYTE TO SCAN
        LA   6,1(6)       ADD 1 TO R6 (COUNT OF ZERO DIGITS+1)
        IC   8,0(3)       GET ANOTHER BYTE OF LEFT
        SR   9,9          CLEAR R9
        SRDL 8,4          UPPER NIBBLE OF BYT IN R8, LWR IN R9
        C    8,ZERO       IF R8 IS ZERO, CONTINUE
        BNE  DONEC        IF NOT, DONE
        BCT  10,CONTC     SCAN NEXT NIBBLE IF MORE LEFT
        B    DONEC1       NO MORE TO SCAN
CONTC    LA   6,1(6)       ADD 1 TO R6 (COUNT OF ZERO DIGITS+1)
        C    9,ZERO       IF R9 IS ZERO, CONTINUE
        BNE  DONEC        IF NOT, DONE
        BCT  10,LOOPC     GET NEXT BYTE IF MORE TO SCAN
        B    DONEC1       NO NEED TO SUBT 1, ALL ZEROES
DONEC    BCTR  6,0          R6 NOW CONTAINS COUNT OF ZERO DIGITS
DONEC1   BCTR  4,0          OFFSET R4 TO PREPARE FOR LOOPD
        LA   10,31        GET NUMBER OF DIGITS TO SCAN
        SR   7,7          CLEAR R7
        SR   8,8          CLEAR R8
LOOPD    LA   4,1(4)       GET ADDRESS OF NEXT BYTE TO SCAN
        LA   7,1(7)       ADD 1 TO R7 (COUNT OF ZERO DIGITS+1)
        IC   8,0(4)       GET ANOTHER BYTE OF RIGHT
        SR   9,9          CLEAR R9
        SRDL 8,4          UPPER NIBBLE OF BYT IN R8, LWR IN R9
        C    8,ZERO       IF R8 IS ZERO, CONTINUE
        BNE  DONED        IF NOT, DONE
        BCT  10,CONTD     CHECK NEXT NIBBLE IF MORE TO SCAN
        B    DONED1       NO MORE TO SCAN
CONTD    LA   7,1(7)       ADD 1 TO R7 (COUNT OF ZERO DIGITS+1)
        C    9,ZERO       IF R9 IS ZERO, CONTINUE
        BNE  DONED        IF NOT, DONE
        BCT  10,LOOPD     GET NEXT BYTE IF MORE TO SCAN
        B    DONED1       NO NEED TO SUBTRACT 1, ALL ZEROES
DONED    BCTR  7,0          R7 NOW CONTAINS COUNT OF ZERO DIGITS
DONED1   LM    3,4,4(1)    RESTORE ADDRESSES OF PARMS
        C    7,SXTEEN     IS DIVISOR BIGGER THAN 8 BYTES
        BL   DIVERR       ERROR IF YES

```

LA 8,31
 SR 8,6
 LA 9,31
 SR 9,7
 SRL 7,1
 LA 6,16
 SR 6,7
 MVC 0(16,2),0(3)
 SR 10,10
 SR 9,8
 BZ DIVCONT
 BP SHFTOP
 LCR 10,9
 SHFTOP SRP 0(16,2),9,5
 DIVCONT NI 15(2),X'FO'
 OI 15(2),X'OC'
 IC 8,15(4)
 LR 9,8
 N 8,=X'FFFFFFFO'
 O 8,=X'000000OC'
 STC 8,15(4)
 CP 0(16,2),0(16,4)
 BL DIVCNT1
 LA 10,1(10)
 DIVCNT1 STC 9,15(4)
 AR 4,7
 LR 8,7
 SLL 7,1
 BCTR 7,0
 SR 7,10
 BM DIVERR
 MVC 0(16,2),0(3)
 BZ DODIV
 SRP 0(16,2),7,5
 B DODIVA
 DODIV SR 7,7
 DODIVA BCTR 6,0
 O 6,=X'000000FO'
 EX 6,DIVISN
 LA 9,16
 SR 9,8
 LR 3,2
 LA 3,15(3)
 AR 2,8
 BCTR 2,0
 MOVLOOP MVC 0(1,3),0(2)
 BCTR 8,0
 BZ NXTLP
 BCTR 2,0
 BCTR 3,0
 B MOVLOOP
 NXTLP BCTR 2,0
 MOVLP1 MVI 0(2),X'00'
 BCTR 9,0
 BZ FINMOV
 BCTR 2,0
 B MOVLP1
 FINMOV ST 7,12(1)
 B DIVRET
 DIVERR LA 3,1
 STC 3,16(1)
 DIVRET RETURN (2,11)
 LOWER DC PL16'-2147483648'

GET MAX DIGITS
 GET NUM DIGS IN DIVIDEND
 GET MAX DIGITS
 GET NUM DIGS IN DIVISOR
 DIVIDE BY 2 => # BYTES OF QUOTIENT
 LOAD R6 WITH 16
 R6 CONTAINS # BYTES IN DIVISOR
 MOVE DIVIDEND TO RESULT FOR TEMP USE
 CLR R10 TO HOLD NUMB DIGS OF RIGHT
 COMP LENGTH(LEFT) WITH LENGTH(RIGHT)
 GOTO DIVCONT IF EQUAL
 GOTO SHFTOP IF LENGTH(L) < LENGTH(R)
 MV #DIGS SHFTD RGHT TO #DIGS IN RES
 SHIFT DIVIDEND FOR COMPARE W/DIVISOR
 CLEAR SIGN OF LEFT
 MAKE SIGN OF LEFT POSITIVE
 GET SIGN OF RIGHT
 SAVE SIGN FOR LATER
 CLEAR SIGN OF RIGHT
 MAKE SIGN OF RIGHT POSITIVE
 STORE SIGN IN RIGHT
 COMPARE RIGHT AND LEFT
 IF LEFT > RIGHT, THEN RESULT WILL
 CONTAIN ONE MORE DIGIT
 REPLACE ACTUAL SIGN INTO RIGHT
 GET OFFSET INTO DIVISOR OF ACTL NUM
 SAVE #BYTES IN QUOTIENT
 GET NUM OF DIGITS + 1 OF QUOTIENT
 GET NUM OF DIGITS OF QUOTIENT
 COMP #DIGS IN QUOTNT TO #DIGS IN RES
 OVERFLOW => GO TO DIVERR
 RESTORE LEFT IN RESULT
 IF EQUAL, THEN PERFORM DIVISION
 SHIFT LEFT TO GET MAX PREC OF RESULT
 GO TO DODIVA
 NO SHIFT TOOK PLACE
 OFFSET #BYTES IN DIVISOR BY ONE
 ADD LENGTH OF DIVIDEND
 PERFORM DIVIDE OPERATION
 MOVE 16 INTO R9
 R9 HAS #BYTES OF ZEROS
 GET ADDRESS OF RESULT INTO R3
 GO TO LAST BYTE
 GET LAST BYTE OF RESULT + 1
 GET LAST BYTE OF RESULT
 MOVE CHARACTER
 SUBTRACT 1 FROM TOTAL TO MOVE
 FINISHED
 GET NEXT BYTE
 GET NEXT BYTE
 MOVE NEXT BYTE
 GET NEXT BYTE
 STORE ZERO
 SUBTRACT ONE FROM R9
 FINISH IF NO MORE TO MOVE
 OTHERWISE, DECREMENT ADDRESS
 MOVE ANOTHER BYTE OF ZEROES
 STORE AMOUNT OF SHIFT INTO PARAM
 GO TO DIVRET
 PUT VALUE 'TRUE' INTO R3
 STORE R3 INTO ERROR

```

UPPER   DC      PL16'2147483648'
POSCON  DC      X'C0000000'
NEGCON  DC      X'D0000000'
POSZCON DC      X'F0000000'
ZERO    DC      F'0'
ONE     DC      F'1'
SXTEEN  DC      F'16'
THYTYTWO DC    F'32'
MULV1A  MP      0(0,2),0(0,3)
MULV2A  MP      0(0,2),0(0,4)
DIVISN  DP      0(0,2),0(0,4)
END     ADASUP

```

C.22 SQL_Char_Pkg Specification

```

with SQL_System; use SQL_System;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Standard;

```

```

package SQL_Char_Pkg
is

```

```

    subtype SQL_Char_Length is natural
        range 1 .. MAXCHLEN;
    subtype SQL_Unpadded_Length is natural
        range 0 .. MAXCHLEN;

```

```

    type SQL_Char_Not_Null is new SQL_Standard.Char;

```

```

    type SQL_Char(Length : SQL_Char_Length) is limited private;

```

```

    function Null_SQL_Char return SQL_Char;
    -- pragma INLINE (Null_SQL_Char);

```

```

    -- the next three functions convert between
    -- null-bearing and non null-bearing-types
    -- Without_Null_Base and With_Null_Base are
    -- inverses (mod. null values)

```

```

    -- see also SQL_Char_Ops generic package below
    function With_Null_Base(Value : SQL_Char_Not_Null)
        return SQL_Char;

```

```

    -- pragma INLINE (With_Null_Base);
    -- Without_Null_Base and Without_Null_Base_Unpadded raise
    -- null_value_error on the null input

```

```

    function Without_Null_Base(Value : SQL_Char) return SQL_Char_Not_Null;
    -- pragma INLINE (Without_Null_Base);

```

```

    -- Without_Null_Unpadded_Base removes trailing blanks from
    -- the input

```

```

    function Without_Null_Unpadded_Base(Value : SQL_Char)
        return SQL_Char_Not_Null;

```

```

    -- pragma INLINE (Without_Null_Unpadded_Base);

```

```

    -- axiom: unpadded Length(x) =
    -- Without_Null_Unpadded_Base(x)'Length
    -- both functions raise null_value_error if x is null

```

```

    -- the next six functions convert between Standard.String
    -- types and the SQL_Char and SQL_Char_Not_Null types

```

```

    function To_String (Value : SQL_Char_Not_Null)
        return String;

```



```

function To_String (Value : SQL_Char)
    return String;
function To_Unpadded_String (Value : SQL_Char_Not_Null)
    return String;
function To_Unpadded_String (Value : SQL_Char)
    return String;
-- pragma INLINE (To_Unpadded_String);
-- this INLINE works for BOTH functions!!
function To_SQL_Char_Not_Null (Value : String)
    return SQL_Char_Not_Null;
function To_SQL_Char (Value : String)
    return SQL_Char;
-- pragma INLINE (To_SQL_Char);

function Unpadded_Length (Value : SQL_Char)
    return SQL_Unpadded_Length;
-- pragma INLINE (Unpadded_Length);

procedure Assign(
    Left : out SQL_Char;
    Right : SQL_Char
);
-- pragma INLINE (Assign);

-- Substring(x,k,m) returns the substring of x starting
-- at position k (relative to 1) with length m.
-- returns null value if x is null
-- raises constraint_error if Start < 1 or Length < 1 or
-- Start + Length - 1 > x.Length
function Substring (Value : SQL_Char;
    Start, Length : SQL_Char_Length)
    return SQL_Char;
-- pragma INLINE (Substring);

-- "&" returns null if either parameter is null;
-- otherwise performs concatenation in the usual way,
-- preserving all blanks.
-- may raise constraint_error implicitly if result is
-- too large (i.e., greater than SQL_Char_Length'Last
function "&" (Left, Right : SQL_Char)
    return SQL_Char;
-- pragma INLINE ("&");

-- Logical Operations --
-- type X type => Boolean_with_unknown --
-- the comparison operators return the boolean value
-- UNKNOWN if either parameter is null; otherwise,
-- the comparison is done in accordance with
-- ANSI X3.135-1986 para 5.11 general rule 5; that is,
-- the shorter of the two string parameters is
-- effectively padded with blanks to be the length of
-- the longer string and a standard Ada comparison is
-- then made
function Equals (Left, Right : SQL_Char) return Boolean_with_Unknown;
-- pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Char)
    return Boolean_with_Unknown;
-- pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Char) return Boolean_with_Unknown;
-- pragma INLINE ("<");
function ">" (Left, Right : SQL_Char) return Boolean_with_Unknown;
-- pragma INLINE (">");
function "<=" (Left, Right : SQL_Char) return Boolean_with_Unknown;

```

```

-- pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Char) return Boolean_with_Unknown;
-- pragma INLINE (">=");

    -- type => boolean --
function Is_Null(Value : SQL_Char) return Boolean;
-- pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Char) return Boolean;
-- pragma INLINE (Not_Null);

-- These functions of class type => boolean
-- equate UNKNOWN with FALSE. That is, they return TRUE
-- only when the function returns TRUE. UNKNOWN and FALSE
-- are mapped to FALSE.
function "=" (Left, Right : SQL_Char) return Boolean;
-- pragma INLINE ("=");
function "<" (Left, Right : SQL_Char) return Boolean;
-- pragma INLINE ("<");
function ">" (Left, Right : SQL_Char) return Boolean;
-- pragma INLINE (">");
function "<=" (Left, Right : SQL_Char) return Boolean;
-- pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Char) return Boolean;
-- pragma INLINE (">=");

-- the purpose of the following generic is to generate
-- conversion functions between a type derived from
-- SQL_Char_Not_Null, which are effectively Ada
-- strings and a type derived from SQL_Char, which
-- mimic the behaviour of SQL strings.
-- the subprogram formal is meant to default; that is,
-- this generic should be instantiated in the scope
-- of an use clause for SQL_Char_Pkg.
generic
type With_Null_Type is limited private;
type Without_Null_Type is array (positive range <>)
of sql_standard.Character_Type;
with function With_Null_Base (Value: SQL_Char_Not_Null)
return With_Null_Type is <>;
with function Without_Null_Base (Value: With_Null_Type)
return SQL_Char_Not_Null is <>;
with function Without_Null_Unpadded_Base (Value: With_Null_Type)
return SQL_Char_Not_Null is <>;
package SQL_Char_Ops is
function With_Null (Value : Without_Null_Type)
return With_Null_Type;
-- pragma INLINE (With_Null);
function Without_Null (Value : With_Null_Type)
return Without_Null_Type;
-- pragma INLINE (Without_Null);
function Without_Null_Unpadded (Value : With_Null_Type)
return Without_Null_Type;
-- pragma INLINE (Without_Null_Unpadded);
end SQL_Char_Ops;

private

type SQL_Char (Length : SQL_Char_Length) is record
Is_Null: Boolean := true;
Unpadded_Length: SQL_Unpadded_Length;
Text: SQL_Char_Not_Null(1 .. Length);
end record;

```

```
end SQL_Char_Pkg;
```

C.23 SQL_Char_Pkg Body

```
With SQL_Exceptions;
with SQL_Standard;
package body SQL_Char_Pkg is

use SQL_Standard.Character_Set; -- literals to be interpreted in
                                -- DBMS native character set

Null_Value_Error : exception renames SQL_Exceptions.Null_Value_Error;

procedure Assign(
  Left : out SQL_Char;
  Right : SQL_Char)
is
begin
  if Right.Is_Null then Left.Is_Null := True;
  else
    Left.Is_Null := False;
    if Left.Length >= Right.Unpadded_Length then
      -- no need to truncate; blank pad
      Left.Unpadded_Length := Right.Unpadded_Length;
      Left.Text := Right.Text(1..Right.Unpadded_Length)
        & SQL_Char_Not_Null'
        (Right.Unpadded_Length + 1 .. Left.Length => ' ');
    else
      -- truncate; may need to strip blanks
      Left.Text(1..Left.Length) := Right.Text(1..Left.Length);
      -- remove trailing blanks in truncated string
      declare
        unpadded_length_ctr : Natural := Left.Length;
      begin
        for i in reverse 1 .. Left.Length loop
          exit when Right.Text(i) /= ' ';
          unpadded_length_ctr := unpadded_length_ctr - 1;
        end loop;
        Left.unpadded_length := unpadded_length_ctr;
      end;
    end if;
  end if;
end Assign;

function With_Null_Base (Value : SQL_Char_Not_Null)
  return SQL_Char is
-- Calculate the Unpadded_Length of the input string
-- without the trailing blanks
-- The input is stored in the output
Unpadded_Length_Ctr : Natural := Value.Length;
subtype Intermed is SQL_Char_Not_Null (1 .. Value.Length); -- allows slices
begin
  for i in reverse Value.First .. Value.Last
  loop
    exit when Value(i) /= ' ';
    Unpadded_Length_Ctr := Unpadded_Length_Ctr - 1;
  end loop;
  return (Length => Value.Length,
    Is_Null => False,
```

```

        Unpadded_Length => Unpadded_Length_Ctr,
        Text => Intermed(Value));
end With_Null_Base;

function Without_Null_Base(Value : SQL_Char) return SQL_Char_Not_Null is
begin
    if Value.Is_Null then
        raise Null_Value_Error;
    else
        return Value.Text;
    end if;
end Without_Null_Base;

function Without_Null_Unpadded_Base(Value : SQL_Char)
return SQL_Char_Not_Null is
begin
    if Value.Is_Null then
        raise Null_Value_Error;
    else
        return (Value.Text(1..Value.Unpadded_Length));
    end if;
end Without_Null_Unpadded_Base;

function Null_SQL_Char return SQL_Char is
    Null_Holder : SQL_Char(1);
begin
    return(Null_Holder); -- relies on default expression for Is_Null
end Null_SQL_Char;

function To_String (Value : SQL_Char_Not_Null)
return String is separate;

function To_String (Value : SQL_Char)
return String is
begin
    if Value.Is_Null then
        raise Null_Value_Error;
    else
        return (To_String(Value.Text));
    end if;
end To_String;

function To_Unpadded_String (Value : SQL_Char_Not_Null)
return String is
begin
    return (To_String(Without_Null_Unpadded_Base(With_Null_Base(Value))));
end To_Unpadded_String;

function To_Unpadded_String (Value : SQL_Char)
return String is
begin
    if Value.Is_Null then
        raise Null_Value_Error;
    else
        return (To_String(Value.Text(1..Value.Unpadded_Length)));
    end if;
end To_Unpadded_String;

function To_SQL_Char_Not_Null (Value : String)
return SQL_Char_Not_Null is separate;

function To_SQL_Char (Value : String)
return SQL_Char is

```

```

-- Calculate the Unpadded_Length of the input string
-- without the trailing blanks
-- The input is stored in the output

Unpadded_Length_Ctr : Natural := Value'Length;
subtype Intermed is SQL_Char_Not_Null (1 .. Value'Length); -- allows slices
begin
  for i in reverse Value'First .. Value'Last
  loop
    exit when Value(i) /= ' ';
    Unpadded_Length_Ctr := Unpadded_Length_Ctr -1;
  end loop;
  return(Length => Value'Length,
         Is_Null => False,
         Unpadded_Length => Unpadded_Length_Ctr,
         Text => Intermed(To_SQL_Char_Not_Null(Value)));
end To_SQL_Char;

function Unpadded_Length (Value : SQL_Char)
  return SQL_Unpadded_Length is
begin
  if Value.Is_Null then
    raise Null_Value_Error;
  else
    return Value.Unpadded_Length;
  end if;
end Unpadded_Length;

function Substring (Value : SQL_Char;
                   Start, Length : SQL_Char_Length)
  return SQL_Char is
begin
  if Value.Is_Null then
    return Null_SQL_Char;
  elsif (Start + Length - 1) > Value.Length then
    -- no need to check Start and Length here to see that
    -- they are > 0
    -- the range constraints on the subtype SQL_Char_Length
    -- will guarantee that a run-time check is made of
    -- these values as they are passed into "Substring"
    raise constraint_error;
  else
    return With_Null_Base(Value.Text(Start .. Start + Length - 1));
  end if;
end Substring;

function "&" (Left, Right : SQL_Char)
  return SQL_Char is
begin
  if Left.Is_Null or else Right.Is_Null then
    return Null_SQL_Char;
  else
    return
      With_Null_Base(Without_Null_Base(Left)
                    & Without_Null_Base(Right));
  end if;
end "&";

function Equals (Left, Right: SQL_Char) return Boolean_With_Unknown is
begin
  if Left.Is_Null or else Right.Is_Null then
    return Unknown;
  else

```

```

        if Left.Text(1..Left.Unpadded_Length) =
            Right.Text(1..Right.Unpadded_Length) then
            return True;
        else
            return False;
        end if;
    end if;
end Equals;

function Not_Equals (Left, Right: SQL_Char) return Boolean_With_Unknown is
begin
    if Left.Is_Null or else Right.Is_Null then
        return Unknown;
    else
        if Left.Text(1..Left.Unpadded_Length) /=
            Right.Text(1..Right.Unpadded_Length) then
            return True;
        else
            return False;
        end if;
    end if;
end Not_Equals;

function ">" (Left, Right: SQL_Char) return Boolean_With_Unknown is
begin
    if Left.Is_Null or else Right.Is_Null then
        return Unknown;
    else
        if Left.Text(1..Left.Unpadded_Length) >
            Right.Text(1..Right.Unpadded_Length) then
            return True;
        else
            return False;
        end if;
    end if;
end;

function ">=" (Left, Right: SQL_Char) return Boolean_With_Unknown is
begin
    if Left.Is_Null or else Right.Is_Null then
        return Unknown;
    else
        if Left.Text(1..Left.Unpadded_Length) >=
            Right.Text(1..Right.Unpadded_Length) then
            return True;
        else
            return False;
        end if;
    end if;
end;

function "<" (Left, Right: SQL_Char) return Boolean_With_Unknown is
begin
    if Left.Is_Null or else Right.Is_Null then
        return Unknown;
    else
        if Left.Text(1..Left.Unpadded_Length) <
            Right.Text(1..Right.Unpadded_Length) then
            return True;
        else
            return False;
        end if;
    end if;
end;

```

```

end;

function "<=" (Left, Right: SQL_Char) return Boolean_With_Unknown is
begin
  if Left.Is_Null or else Right.Is_Null then
    return Unknown;
  else
    if Left.Text(1..Left.Unpadded_Length) <=
      Right.Text(1..Right.Unpadded_Length) then
      return True;
    else
      return False;
    end if;
  end if;
end;

function Is_Null(Value : SQL_Char) return Boolean is
begin
  return Value.Is_Null;
end Is_Null;

function Not_Null(Value : SQL_Char) return Boolean is
begin
  return not Value.Is_Null;
end Not_Null;

function "=" (Left, Right: SQL_Char) return Boolean is
begin
  if Left.Is_Null or else Right.Is_Null then
    return FALSE;
  else
    if Left.Text(1..Left.Unpadded_Length) =
      Right.Text(1..Right.Unpadded_Length) then
      return True;
    else
      return False;
    end if;
  end if;
end "=";

function "<" (Left, Right: SQL_Char) return Boolean is
begin
  if Left.Is_Null or else Right.Is_Null then
    return FALSE;
  else
    if Left.Text(1..Left.Unpadded_Length) <
      Right.Text(1..Right.Unpadded_Length) then
      return True;
    else
      return False;
    end if;
  end if;
end "<";

function ">" (Left, Right: SQL_Char) return Boolean is
begin
  if Left.Is_Null or else Right.Is_Null then
    return FALSE;
  else
    if Left.Text(1..Left.Unpadded_Length) >
      Right.Text(1..Right.Unpadded_Length) then
      return True;
    else

```

```

        return False;
    end if;
end ">";

function "<=" (Left, Right: SQL_Char) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        if Left.Text(1..Left.Unpadded_Length) <=
            Right.Text(1..Right.Unpadded_Length) then
            return True;
        else
            return False;
        end if;
    end if;
end "<=";

function ">=" (Left, Right: SQL_Char) return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return FALSE;
    else
        if Left.Text(1..Left.Unpadded_Length) >=
            Right.Text(1..Right.Unpadded_Length) then
            return True;
        else
            return False;
        end if;
    end if;
end ">=";

package body SQL_Char_Ops is
    function With_Null (Value : Without_Null_Type)
        return With_Null_Type is
    begin
        return With_Null_Base(SQL_Char_Not_Null(Value));
    end With_Null;

    function Without_Null (Value : With_Null_Type)
        return Without_Null_Type is
    begin
        return Without_Null_Type(
            SQL_Char_Not_Null'(Without_Null_Base(Value)));
    end Without_Null;

    function Without_Null_Unpadded (Value : With_Null_Type)
        return Without_Null_Type is
    begin
        return Without_Null_Type(
            SQL_Char_Not_Null'(Without_Null_Unpadded_Base(Value)));
    end Without_Null_Unpadded;

end SQL_Char_Ops;

end SQL_Char_Pkg;

```


C.24 Subunit To_String

```
-- assuming an ascii host character set
-- that is SQL_Standard.Character_Type is Standard.Character
separate (SQL_Char_Pkg)
function To_String (Value : SQL_Char_Not_Null)
  return String is
begin
  return (String(Value));
end To_String;
```

C.25 Subunit To_SQL_Char_Not_Null

```
-- assuming an ascii host character set
-- that is SQL_Standard.Character_Type is Standard.Character
separate (SQL_Char_Pkg)
function To_SQL_Char_Not_Null (Value : String)
  return SQL_Char_Not_Null is
begin
  return (SQL_Char_Not_Null(Value));
end To_SQL_Char_Not_Null;
```

C.26 SQL_Enumeration_Pkg Specification

```
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Char_Pkg; use SQL_Char_Pkg;
generic
  type SQL_Enumeration_Not_Null is (<>);
package SQL_Enumeration_Pkg
  is
    ---- Possibly Null Enumeration ----
    type SQL_Enumeration is limited private;

    function Null_SQL_Enumeration return SQL_Enumeration;
    -- pragma INLINE (Null_SQL_Enumeration);

    -- this pair of functions convert between the
    -- null-bearing and non-null-bearing types.
    function Without_Null(Value : in SQL_Enumeration)
      return SQL_Enumeration_Not_Null;
    -- pragma INLINE (Without_Null);
    -- With_Null raises Null_Value_Error if the input
    -- value is null
    function With_Null(Value : in SQL_Enumeration_Not_Null)
      return SQL_Enumeration;
    -- pragma INLINE (With_Null);

    procedure Assign (
      Left : in out SQL_Enumeration; Right : in SQL_Enumeration);
    -- pragma INLINE (Assign);

    -- Logical Operations --
    -- type X type => Boolean_with_unknown --
    -- these functions implement three valued logic
    -- if either input is the null value, the functions
    -- return the truth value UNKNOWN; otherwise they
    -- perform the indicated comparison.
```

```

-- these functions raise no exceptions
function Equals (Left, Right : SQL_Enumeration)
    return Boolean_with_Unknown;
function Not_Equals (Left, Right : SQL_Enumeration)
    return Boolean_with_Unknown;
-- pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Enumeration) return Boolean_with_Unknown;
function ">" (Left, Right : SQL_Enumeration) return Boolean_with_Unknown;
function "<=" (Left, Right : SQL_Enumeration) return Boolean_with_Unknown;
function ">=" (Left, Right : SQL_Enumeration) return Boolean_with_Unknown;

```

```

-- type => boolean --
function Is_Null (Value : SQL_Enumeration) return Boolean;
-- pragma INLINE (Is_Null);
function Not_Null (Value : SQL_Enumeration) return Boolean;
-- pragma INLINE (Not_Null);
function "=" (Left, Right : SQL_Enumeration) return Boolean;
-- pragma INLINE ("=");
function "<" (Left, Right : SQL_Enumeration) return Boolean;
-- pragma INLINE ("<");
function ">" (Left, Right : SQL_Enumeration) return Boolean;
-- pragma INLINE (">");
function "<=" (Left, Right : SQL_Enumeration) return Boolean;
-- pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Enumeration) return Boolean;
-- pragma INLINE (">=");

```

```

-- the following six functions mimic the
-- 'Pred, 'Succ, 'Image, 'Pos, 'Val, and 'Value
-- attributes of the SQL_Enumeration_Not_Null type, passed
-- in, for the associated SQL_Enumeration (null) type
-- they all raise the Null_Value_Error exception if a null
-- value is passed in
-- Pred raises the Constraint_Error exception if the value
-- passed in is equal to SQL_Enumeration_Not_Null'Last
-- Succ raises the Constraint_Error exception if the value
-- passed in is equal to SQL_Enumeration_Not_Null'First
-- Val raises the Constraint_Error exception if the value passed
-- in is not in the range P'POS(P'FIRST)..P'POS(P'LAST) for type P
-- Value raises the Constraint_Error exception if the sequence of
-- characters passed in does not have the syntax of an enumeration
-- literal for the instantiated enumeration type
function Pred (Value : in SQL_Enumeration) return SQL_Enumeration;
-- pragma INLINE (Pred);
function Succ (Value : in SQL_Enumeration) return SQL_Enumeration;
-- pragma INLINE (Succ);
function Pos (Value : in SQL_Enumeration) return Integer;
-- pragma INLINE (Pos);
function Image (Value : in SQL_Enumeration) return SQL_Char;
function Image (Value : in SQL_Enumeration_Not_Null)
    return SQL_Char_Not_Null;
-- pragma INLINE (Image);
function Val (Value : in Integer) return SQL_Enumeration;
-- pragma INLINE (Val);
function Value (Value : in SQL_Char) return SQL_Enumeration;
function Value (Value : in SQL_Char_Not_Null)
    return SQL_Enumeration_Not_Null;
-- pragma INLINE (Value);

```

private

```

type SQL_Enumeration is record

```

```

        Is_Null: Boolean := true;
        Value: SQL_Enumeration_Not_Null;
    end record;

end SQL_Enumeration_Pkg;

```

C.27 SQL_Enumeration_Pkg Body

```

With SQL_Exceptions;
package body SQL_Enumeration_Pkg
is
    Null_Value_Error : exception renames SQL_Exceptions.Null_Value_Error;

    function Null_SQL_Enumeration return SQL_Enumeration is
        Null_Holder : SQL_Enumeration;
    begin
        return Null_Holder;
    end Null_SQL_Enumeration;

    function Without_Null(Value : in SQL_Enumeration)
        return SQL_Enumeration_Not_Null is
    begin
        if Value.Is_Null then
            raise Null_Value_Error;
        else
            return Value.Value;
        end if;
    end Without_Null;

    function With_Null(Value : in SQL_Enumeration_Not_Null)
        return SQL_Enumeration is
    begin
        return (Is_Null => false,
                Value => Value);
    end With_Null;

    procedure Assign (Left : in out SQL_Enumeration;
                     Right : in SQL_Enumeration) is
    begin
        Left := Right;
    end Assign;

    function Equals (Left, Right : SQL_Enumeration)
        return Boolean_With_Unknown is
    begin
        if Left.Is_Null or else Right.Is_Null then
            return Unknown;
        elsif Left.Value = Right.Value then
            return True;
        else
            return False;
        end if;
    end Equals;

    function Not_Equals (Left, Right : SQL_Enumeration)
        return Boolean_With_Unknown is
    begin
        if Left.Is_Null or else Right.Is_Null then
            return Unknown;

```

```

        elsif Left.Value /= Right.Value then
            return True;
        else
            return False;
        end if;
    end Not_Equals;

function "<" (Left, Right : SQL_Enumeration)
    return Boolean_With_Unknown is
begin
    if Left.Is_Null or else Right.Is_Null then
        return Unknown;
    elsif Left.Value < Right.Value then
        return True;
    else
        return False;
    end if;
end "<";

function ">" (Left, Right : SQL_Enumeration)
    return Boolean_With_Unknown is
begin
    if Left.Is_Null or else Right.Is_Null then
        return Unknown;
    elsif Left.Value > Right.Value then
        return True;
    else
        return False;
    end if;
end ">";

function "<=" (Left, Right : SQL_Enumeration)
    return Boolean_With_Unknown is
begin
    if Left.Is_Null or else Right.Is_Null then
        return Unknown;
    elsif Left.Value <= Right.Value then
        return True;
    else
        return False;
    end if;
end "<=";

function ">=" (Left, Right : SQL_Enumeration)
    return Boolean_With_Unknown is
begin
    if Left.Is_Null or else Right.Is_Null then
        return Unknown;
    elsif Left.Value >= Right.Value then
        return True;
    else
        return False;
    end if;
end ">=";

function Is_Null (Value : SQL_Enumeration)
    return Boolean is
begin
    return Value.Is_Null;
end Is_Null;

function Not_Null (Value : SQL_Enumeration)
    return Boolean is

```

```

begin
    return not Value.Is_Null;
and Not_Null;

function "=" (Left, Right : SQL_Enumeration)
    return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return False;
    elsif Left.Value = Right.Value then
        return True;
    else
        return False;
    end if;
end "=";

function "<" (Left, Right : SQL_Enumeration)
    return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return False;
    elsif Left.Value < Right.Value then
        return True;
    else
        return False;
    end if;
end "<";

function ">" (Left, Right : SQL_Enumeration)
    return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return False;
    elsif Left.Value > Right.Value then
        return True;
    else
        return False;
    end if;
end ">";

function "<=" (Left, Right : SQL_Enumeration)
    return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return False;
    elsif Left.Value <= Right.Value then
        return True;
    else
        return False;
    end if;
end "<=";

function ">=" (Left, Right : SQL_Enumeration)
    return Boolean is
begin
    if Left.Is_Null or else Right.Is_Null then
        return False;
    elsif Left.Value >= Right.Value then
        return True;
    else
        return False;
    end if;
end ">=";

```

```

function Pred (Value : in SQL_Enumeration)
  return SQL_Enumeration is
begin
  if Value.Is_Null then
    return Null_SQL_Enumeration;
  else
    return (With_Null(SQL_Enumeration_Not_Null'Pred(Value.Value)));
  end if;
end Pred;

function Succ (Value : in SQL_Enumeration)
  return SQL_Enumeration is
begin
  if Value.Is_Null then
    return Null_SQL_Enumeration;
  else
    return (With_Null(SQL_Enumeration_Not_Null'Succ(Value.Value)));
  end if;
end Succ;

function Pos (Value : in SQL_Enumeration) return Integer is
begin
  if Value.Is_Null then
    raise Null_Value_Error;
  else
    return SQL_Enumeration_Not_Null'Pos(Value.Value);
  end if;
end Pos;

function Image (Value : in SQL_Enumeration_Not_Null)
  return SQL_Char_Not_Null is
begin
  return To_SQL_Char_Not_Null(
    SQL_Enumeration_Not_Null'Image(Value));
end Image;

function Image (Value : in SQL_Enumeration)
  return SQL_Char is
begin
  if Value.Is_Null then
    raise Null_Value_Error;
  else
    return To_SQL_Char(SQL_Enumeration_Not_Null'Image(Value.Value));
  end if;
end Image;

function Val (Value : in Integer) return SQL_Enumeration is
begin
  return (With_Null(SQL_Enumeration_Not_Null'Val(Value)));
end Val;

function Value (Value : in SQL_Char_Not_Null)
  return SQL_Enumeration_Not_Null is
begin
  return (SQL_Enumeration_Not_Null'Value(To_String(Value)));
end Value;

function Value (Value : in SQL_Char)
  return SQL_Enumeration is
begin
  If Is_Null(Value) then
    return Null_SQL_Enumeration;
  else

```

```

        return With_Null(SQL_Enumeration_Not_Null'Value(
            To_String(Value)));
    end if;
end Value;

end SQL_Enumeration_Pkg;

```

C.28 SQL_Database_Error_Pkg Specification

```

package SQL_Database_Error_Pkg is

    -- The following procedure must be present in every version of
    -- SQL_Database_Error_Pkg. It's purpose is to perform standard
    -- processing of unexpected exceptional conditions. It should not
    -- attempt error recover.

    procedure Process_Database_Error;

end SQL_Database_Error_Pkg;

```

C.29 SQL_Database_Error_Pkg Body

```

with Text_IO, SQL_Communications_Pkg, SQL_Base_Types_Pkg;
use Text_IO, SQL_Communications_Pkg, SQL_Base_Types_Pkg;
package body SQL_Database_Error_Pkg is

    procedure Process_Database_Error is

    begin

        -- Procedure Process_Database_Error is called in response
        -- to an unexpected database exception (an error incident).
        -- The procedure may be modified per
        -- the needs of the Abstract Interface developer

        -- This is a minimal implementation.
        -- Get a descriptive error message from the DBMS
        -- (through the package SQL_Communications_Pkg)
        -- and display it on standard output.

        put_line (To_String(SQL_Char_Not_Null(SQL_Database_Error_Message)));

    end Process_Database_Error;

end SQL_Database_Error_Pkg;

```

C.30 SQL_Date_Pkg Specification

```

with SQL_Standard;
with Calendar; use Calendar;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Char_Pkg; use SQL_Char_Pkg;
package SQL_Date_Pkg
is

    type precision is range 0..10;

```

```

type SQL_Datetime_Field is (year, month, day,
                             hour, minute, second, fraction);
type SQL_Date_Not_Null is new SQL_Char_Not_Null;
type SQL_Date(From      : SQL_Datetime_Field;
              To        : SQL_Datetime_Field;
              Fractional : precision) is limited private;
type SQL_Interval(From      : SQL_Datetime_Field;
                 Leading    : precision;
                 To        : SQL_Datetime_Field;
                 Fractional : precision) is limited private;

function Null_SQL_Date return SQL_Date;
-- pragma INLINE (Null_SQL_Date);

function Null_SQL_Interval return SQL_Interval;
-- pragma INLINE (Null_SQL_Interval);

-- these functions return the not-null portion of the null-bearing type
function Without_Null_Base(Value : SQL_Date) return SQL_Date_Not_Null;
function Without_Null_Base(Value : SQL_Interval) return SQL_Date_Not_Null;
-- pragma INLINE (Without_Null_Base);

-- this function returns an object of the standard.duration type, after
-- converting to it from the input object of type SQL_Interval
function To_Duration (Value : SQL_Interval) return duration;
-- pragma INLINE (To_Duration);

-- this function returns an object of the calendar.time type, after
-- converting to it from the input object of type SQL_Date
function To_Time (Value : SQL_Date) return time;
-- pragma INLINE (To_Time);

-- these procedures parse the input of type SQL_Date_Not_Null, and assign
-- the datetime and interval field values to the objects of type
-- SQL_Date and SQL_Interval, using discriminants that it determines are
-- the correct ones for the object.  If these discriminants differ from
-- the ones supplied in the abstract domain for the object when it was
-- declared, a constraint_error will be raised.
procedure Parse_and_Assign_Base(Left: in out SQL_Date;
                               Right : SQL_Date_Not_Null);
procedure Parse_and_Assign_Base(Left: in out SQL_Interval;
                               Right : SQL_Date_Not_Null);
-- pragma INLINE (Parse_and_Assign);

-- this function accepts input of type standard.duration, and
-- returns an object of type SQL_Interval whose not-null portion
-- has the correct SQL "interval" value specification format,
-- (FROM => day, LEADING => 2, TO => fraction, FRACTIONAL => 3)
function To_SQL_Interval (Value : duration) return SQL_Interval;
-- pragma INLINE (To_SQL_Interval);

-- this function accepts input of type standard.time, and
-- returns an object of type SQL_Date whose not-null portion
-- has the correct SQL "datetime" value specification format
function To_SQL_Date (Value : time) return SQL_Date;
-- pragma INLINE (To_SQL_Date);

-- the assign procedure assigns Right to Left
procedure Assign(Left : in out SQL_Date; Right : SQL_Date);
procedure Assign(Left : in out SQL_Interval; Right : SQL_Interval);
-- pragma INLINE (Assign);

-- the following three functions implement unary "+", "-", "abs"

```



```

-- for the SQL_Interval type
function "+"(Right : SQL_Interval) return SQL_Interval;
function "-"(Right : SQL_Interval) return SQL_Interval;
function "abs"(Right : SQL_Interval) return SQL_Interval;
-- pragma INLINE ("abs");

-- the following functions implement three valued
-- arithmetic
-- if either input to any of these functions is null
-- the function returns the null value; otherwise
-- they perform the indicated operation
-- these functions raise no exceptions
function "+"(Left, Right : SQL_Interval) return SQL_Interval;
function Plus(Left : SQL_Interval; Right : SQL_Date) return SQL_Date;
function Plus(Left : SQL_Date; Right : SQL_Interval) return SQL_Date;
-- pragma INLINE ("+");
function "-"(Left, Right : SQL_Interval) return SQL_Interval;
function Minus(Left, Right : SQL_Date) return SQL_Interval;
function Minus(Left : SQL_Date; Right : SQL_Interval) return SQL_Date;
-- pragma INLINE ("-");
function "*" (Left : SQL_Interval; Right : integer) return SQL_Interval;
-- pragma INLINE ("*");
function "/"(Left : SQL_Interval; Right : integer) return SQL_Interval;
-- pragma INLINE ("/");

-- Logical Operations --
-- type X type => Boolean_with_unknown --
-- these functions implement three valued logic
-- if either input is the null value, the functions
-- return the truth value UNKNOWN; otherwise they
-- perform the indicated comparison.
-- these functions raise no exceptions
function Equals (Left, Right : SQL_Date) return Boolean_with_Unknown;
function Equals (Left, Right : SQL_Interval) return Boolean_with_Unknown;
-- pragma INLINE (Equals);
function Not_Equals (Left, Right : SQL_Date)
    return Boolean_with_Unknown;
function Not_Equals (Left, Right : SQL_Interval)
    return Boolean_with_Unknown;
-- pragma INLINE (Not_Equals);
function "<" (Left, Right : SQL_Date) return Boolean_with_Unknown;
function "<" (Left, Right : SQL_Interval) return Boolean_with_Unknown;
-- pragma INLINE ("<");
function ">" (Left, Right : SQL_Date) return Boolean_with_Unknown;
function ">" (Left, Right : SQL_Interval) return Boolean_with_Unknown;
-- pragma INLINE (">");
function "<=" (Left, Right : SQL_Date) return Boolean_with_Unknown;
function "<=" (Left, Right : SQL_Interval) return Boolean_with_Unknown;
-- pragma INLINE ("<=");
function ">=" (Left, Right : SQL_Date) return Boolean_with_Unknown;
function ">=" (Left, Right : SQL_Interval) return Boolean_with_Unknown;
-- pragma INLINE (">=");

-- type => boolean --
function Is_Null(Value : SQL_Date) return Boolean;
function Is_Null(Value : SQL_Interval) return Boolean;
-- pragma INLINE (Is_Null);
function Not_Null(Value : SQL_Date) return Boolean;
function Not_Null(Value : SQL_Interval) return Boolean;
-- pragma INLINE (Not_Null);
function Is_Year_Month(Value : SQL_Interval) return Boolean;
-- pragma INLINE (Is_Year_Month);
function Is_Day_Time(Value : SQL_Interval) return Boolean;

```

```

-- pragma INLINE(Is_Day_Time);
function Not_Year_Month(Value : SQL_Interval) return Boolean;
-- pragma INLINE(Not_Year_Month);
function Not_Day_Time(Value : SQL_Interval) return Boolean;
-- pragma INLINE (Not_Day_Time);

-- the procedure Current returns the current system Datetime, using
-- the precision of the input variable
procedure Current (Value : in out SQL_Date);
-- pragma INLINE(Current);
-- the procedure Extend returns the value of the Right input object with
-- the datetime qualifier of the Left object, if a valid datetime
-- value is generated by the extension process
procedure Extend (Value : in out SQL_Date);
-- pragma INLINE(Extend);

-- this generic is instantiated once for every abstract
-- SQL_Date domain, and once for every abstract SQL_Interval
-- domain, based on the type SQL_Date_Not_Null.
-- the two subprogram formal parameters are meant to
-- default to the programs declared above.
-- that is, the package should be instantiated in the
-- scope of a use clause for SQL_Date_Pkg.
-- the two actual types together form the abstract
-- domain.
-- the purpose of the generic is to create functions
-- which convert between the two actual types
-- the bodies of these subprograms are calls to
-- subprograms declared above and passed as defaults to
-- the generic.
generic
type With_Null_Type is limited private;
type Without_Null_Type is array (positive range <>)
of SQL_Standard.Character_Type;
with procedure Parse_and_Assign_Base
(Left : in out With_Null_Type; Right : SQL_Date_Not_Null) is <>;
with function Without_Null_Base(Value : With_Null_Type)
return SQL_Date_Not_Null is <>;
package SQL_Date_Ops is
procedure Parse_and_Assign (Left : With_Null_Type;
Right : Without_Null_Type);
-- pragma INLINE (Parse_and_Assign);
function Without_Null (Value : With_Null_Type)
return Without_Null_Type;
-- pragma INLINE (Without_Null);
end SQL_Date_Ops;

generic
type With_Null_Date_Type is limited private;
type With_Null_Interval_Type is limited private;
with function Plus (Left : With_Null_Date_Type; Right : SQL_Interval)
return With_Null_Date_Type is <>;
with function Plus (Left : SQL_Interval; Right : With_Null_Date_Type)
return With_Null_Date_Type is <>;
with function Minus (Left : With_Null_Date_Type; Right : SQL_Interval)
return With_Null_Date_Type is <>;
with function Minus (Left, Right : With_Null_Date_Type)
return SQL_Interval is <>;
package SQL_Date_Interval_Ops is
function "+" (Left : With_Null_Date_Type; Right : With_Null_Interval_Type)
return With_Null_Date_Type;
function "+" (Left : With_Null_Interval_Type; Right : With_Null_Date_Type)
return With_Null_Date_Type;

```

```

function "-" (Left : With_Null_Date_Type; Right : With_Null_Interval_Type)
    return With_Null_Date_Type;
function "-" (Left, Right : With_Null_Date_Type)
    return With_Null_Interval_Type;
end SQL_Date_Interval_Ops;

private

type SQL_year_number    is range 1600..9999;
type SQL_month_number   is range 1..12;
type SQL_day_number     is range 1..31;
type SQL_hour_number    is range 0..23;
type SQL_minute_number  is range 0..59;
type SQL_second_number  is range 0..59;
type SQL_fraction_number is range 0..(2**31)-1;
type SQL_interval_number is range -(2**31)..(2**31)-1;

type SQL_Date(From      : SQL_Datetime_Field;
               To        : SQL_Datetime_Field;
               Fractional : precision)
is record
    Is_Null : Boolean := true;
    year    : SQL_year_number;
    month   : SQL_month_number;
    day     : SQL_day_number;
    hour    : SQL_hour_number;
    minute  : SQL_minute_number;
    second  : SQL_second_number;
    fraction : SQL_fraction_number;
end record;

type SQL_Interval(From      : SQL_Datetime_Field;
                  Leading    : precision;
                  To        : SQL_Datetime_Field;
                  Fractional : precision)
is record
    Is_Null      : boolean := True;
    Is_Year_Month : boolean := True;
    years        : SQL_interval_number;
    months       : SQL_interval_number;
    days         : SQL_interval_number;
    minutes      : SQL_interval_number;
    seconds      : SQL_interval_number;
    fraction     : SQL_interval_number;
end record;

end SQL_Date_Pkg;

```

C.31 INGRES_Date_Pkg Specification

```

with SQL_Standard;
with SQL_System; use SQL_System;
with Calendar; use Calendar;
with SQL_Boolean_Pkg; use SQL_Boolean_Pkg;
with SQL_Char_Pkg; use SQL_Char_Pkg;
package INGRES_Date_Pkg
    is

        type INGRES_Date_Not_Null is new SQL_Char_Not_Null;
        ---- Possibly Null Datetime ----

```

```

type INGRES_Date Format is (Datetime, Interval, Unknown);
type INGRES_Date(Format : INGRES_Date_Format := Unknown)
  is limited private;

function Null_INGRES_Date return INGRES_Date;
-- pragma INLINE (Null_INGRES_Date);

-- this function accepts input of type INGRES_Date_Not_Null, and
-- returns an object whose not-null portion is the input
function With_Null_Base(Value : INGRES_Date_Not_Null)
  return INGRES_Date;
-- pragma INLINE (With_Null_Base);

-- this function returns the not-null portion of the null-bearing type
function Without_Null_Base(Value : INGRES_Date)
  return INGRES_Date_Not_Null;
-- pragma INLINE (Without_Null_Base);

-- this function returns the not-null portion of the null-bearing type
-- this function differs from Without_Null_Base in that the output
-- is extended to include all fields,
-- even if they contain a value of zero
-- INGRES may output a date in a format
-- that is unacceptable as INGRES input.
-- Therefore this function extends the output format into an acceptable
-- INGRES input format, and should be used when interacting with INGRES
function Without_Null_DBMS_Base(Value : INGRES_Date)
  return INGRES_Date_Not_Null;
-- pragma INLINE (Without_Null_DBMS_Base);

-- this function raises constraint_error if the object of type
-- INGRES_Date_Not_Null is not in the correct INGRES "interval" format
-- of the INGRES date data type
function To_Duration (Value : INGRES_Date) return duration;
-- pragma INLINE (To_Duration);

-- this function raises constraint_error if the object of type
-- INGRES_Date_Not_Null is not in the correct INGRES "datetime" format
-- of the INGRES date data type
function To_Time (Value : INGRES_Date) return time;
-- pragma INLINE (To_Time);

-- this function accepts input of type standard.duration, and
-- returns an object whose not-null portion has the correct INGRES
-- "interval" format of the INGRES date data type
function To_INGRES_Date (Value : duration) return INGRES_Date;

-- this function accepts input of type standard.time, and
-- returns an object whose not-null portion has the INGRES "datetime"
-- format of the INGRES date data type
function To_INGRES_Date (Value : time) return INGRES_Date;
-- pragma INLINE (To_INGRES_Date);

procedure Assign(Left : in out INGRES_Date; Right : INGRES_Date);
-- pragma INLINE (Assign);

-- the following three functions implement unary "+", "-", "abs"
function "+"(Right : INGRES_Date) return INGRES_Date;

function "-"(Right : INGRES_Date) return INGRES_Date;

function "abs"(Right : INGRES_Date) return INGRES_Date;
-- pragma INLINE ("abs");

```

```

-- the following functions implement three valued
-- arithmetic
-- if either input to any of these functions is null
-- the function returns the null value; otherwise
-- they perform the indicated operation
-- these functions raise no exceptions
function "+"(Left, Right : INGRES_Date) return INGRES_Date;
-- pragma INLINE ("+");
function "-"(Left, Right : INGRES_Date) return INGRES_Date;
-- pragma INLINE ("-");

-- Logical Operations --
-- type X type => Boolean_with_unknown --
-- these functions implement three valued logic
-- if either input is the null value, the functions
-- return the truth value UNKNOWN; otherwise they
-- perform the indicated comparison.
-- these functions raise no exceptions
function Equals (Left, Right : INGRES_Date) return Boolean_with_Unknown;
function Not_Equals (Left, Right : INGRES_Date)
    return Boolean_with_Unknown;
function "<" (Left, Right : INGRES_Date) return Boolean_with_Unknown;
function ">" (Left, Right : INGRES_Date) return Boolean_with_Unknown;
function "<=" (Left, Right : INGRES_Date) return Boolean_with_Unknown;
function ">=" (Left, Right : INGRES_Date) return Boolean_with_Unknown;

-- type => boolean --
function Is_Null(Value : INGRES_Date) return Boolean;
-- pragma INLINE (Is_Null);
function Not_Null(Value : INGRES_Date) return Boolean;
-- pragma INLINE (Not_Null);
function Equals (Left, Right : INGRES_Date) return Boolean;
-- pragma INLINE (Equals);
function Not_Equals (Left, Right : INGRES_Date)
    return Boolean;
-- pragma INLINE (Not_Equals);
function "<" (Left, Right : INGRES_Date) return Boolean;
-- pragma INLINE ("<");
function ">" (Left, Right : INGRES_Date) return Boolean;
-- pragma INLINE (">");
function "<=" (Left, Right : INGRES_Date) return Boolean;
-- pragma INLINE ("<=");
function ">=" (Left, Right : INGRES_Date) return Boolean;
-- pragma INLINE (">=");

-- this generic is instantiated once for every abstract
-- domain based on the type INGRES_Date_Not_Null.
-- the two subprogram formal parameters are meant to
-- default to the programs declared above.
-- that is, the package should be instantiated in the
-- scope of a use clause for INGRES_Date_Pkg.
-- the two actual types together form the abstract
-- domain.
-- the purpose of the generic is to create functions
-- which convert between the two actual types
-- the bodies of these subprograms are calls to
-- subprograms declared above and passed as defaults to
-- the generic.
generic
type With_Null_Type is limited private;
type Without_Null_Type is array (positive range <>)
    of SQL_Standard.Character_Type;
with function With_Null_Base(Value : INGRES_Date_Not_Null)

```

```

    return With_Null_Type is <>;
with function Without_Null_Base(Value : With_Null_Type)
    return INGRES_Date_Not_Null is <>;
with function Without_Null_DBMS_Base(Value : With_Null_Type)
    return INGRES_Date_Not_Null is <>;
package INGRES_Date_Ops is
    function With_Null (Value : Without_Null_Type)
        return With_Null_Type;
    -- pragma INLINE (With_Null);
    function Without_Null (Value : With_Null_Type)
        return Without_Null_Type;
    -- pragma INLINE (Without_Null);
    function Without_Null_DBMS (Value : With_Null_Type)
        return Without_Null_Type;
    -- pragma INLINE (Without_Null_DBMS);
end INGRES_Date_Ops;

private

type INGRES_year_number    is range 1582..2382;
type INGRES_month_number  is range 1..12;
type INGRES_day_number    is range 1..31;
type INGRES_hour_number   is range 0..23;
type INGRES_minute_number is range 0..59;
type INGRES_second_number is range 0..59;
type years_number         is range -800..800;
type months_number        is range -(800*12)..(800*12);
type days_number          is range -(292200)..(292200); -- 800 * 365.25
type hours_number         is range -(292200*24)..(292200*24);
type minutes_number       is range -(292200*24*60)..(292200*24*60);
type seconds_number       is range -(2**31)..(2**31)-1;

type INGRES_Date (Format : INGRES_Date_Format := Unknown) is record
    Is_Null: Boolean := true;
    case Format is
        when Datetime =>
            year    : INGRES_year_number;
            month   : INGRES_month_number;
            day     : INGRES_day_number;
            hour    : INGRES_hour_number;
            minute  : INGRES_minute_number;
            second  : INGRES_second_number;
        when Interval =>
            years   : years_number;
            months  : months_number;
            days    : days_number;
            hours   : hours_number;
            minutes : minutes_number;
            seconds : seconds_number;
        when Unknown =>
            null;
    end case;
end record;

end INGRES_Date_Pkg;

```

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS NONE	
2a. SECURITY CLASSIFICATION AUTHORITY N/A		3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE DISTRIBUTION UNLIMITED	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A			
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CMU/SEI-89-TR-16		5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-89-24	
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST.	6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE	
6c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213		7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SFI JOINT PROGRAM OFFICE	8b. OFFICE SYMBOL (If applicable) ESD/XRS1	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F1962885C0003	
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213		10. SOURCE OF FUNDING NOS.	
		PROGRAM ELEMENT NO. 63752F	PROJECT NO. N/A
11. TITLE (Include Security Classification) GUIDELINES FOR THE USE OF THE SAME			
12. PERSONAL AUTHOR(S) Marc H. Graham			
13a. TYPE OF REPORT FINAL	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) May 1989	15. PAGE COUNT 249
16. SUPPLEMENTARY NOTATION			
17. COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Ada SQL (structured language query) data base SAME (SQL Ada Module Extensions) DBMS (data base management system)	
FIELD	GROUP		
	SUB. GR.		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) These guidelines describe the Structured Query Language (SQL) Ada Module Extensions, or SAME, a method for the construction of Ada applications that access database management systems whose data manipulation language is SQL. As its name implies, the SAME extends the module language defined in the ANSI SQL standard to fit the needs of Ada. The defining characteristic of the use of the module language is that the SQL statements appear together, physically separated from the Ada application, in an object called the module. The Ada application accesses the module through procedure calls. The primary audience for this document consists of application developers and technicians creating Ada applications for SQL database management systems. The document contains a complete description of the SAME, including its motivation.			
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> OTIC USERS <input checked="" type="checkbox"/>		21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED DISTRIBUTION	
2a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER		22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630	22c. OFFICE SYMBOL SEI JPO