Technical Report

CMU/SEI-91-TR-12
ESD-TR-91-12

# Notes on Applications of the SQL Ada Module Description Language (SAMeDL)

Gary Chastek
Marc H. Graham
Gregory Zelesnik

June 1991

Technical Report
CMU/SEI-91-TR-12
ESD-TR-91-12
June 1991

# Notes on Applications of the SQL Ada Module Description Language (SAMeDL)

Gary Chastek
Marc H. Graham
Gregory Zelesnik

Binding of Ada and SQL Project



**Software Engineering Institute**
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

# Notes on Applications of the SQL Ada Module Description Language (SAMeDL)

**Abstract**: The SQL Ada Module Description Language (SAMeDL) is a language for describing information services to be provided to Ada application programs by SQL database management systems. This report shows how the SAMeDL can be adapted and extended to provide services to applications needing advanced features (e.g., dynamic SQL), or using non-ANSI standard data types (decimal, date) or having other unusual requirements. It also contains short descriptions of some implementation details.

# 1. Introduction

The Ada SQL Binding Project at the SEI has been working on the issues involved in Ada database management system (DBMS) application programming since November 1987. Most of our work was done with the able assistance, help and advice of a Design Committee, a public group of experts from industry and academia.[1] The products of our labors are a method of construction or architecture for Ada DBMS applications known as the SQL Ada Module Extensions, or SAME, and a language used to implement the architecture, the SQL Ada Module Description Language, or SAMeDL. The method is described in SEI technical report *Guidelines for the Use of the SAME* [11]. The language is defined in *The SQL Ada Module Description Language SAMeDL* [18] and explained in *Rationale for the SQL Ada Module Description Language* [9].

As we've worked on this technology, we've confronted a number of issues that, although not important to the "average" DBMS application are important to a significant minority of the DBMS community. This report is our response to those needs.

This next four chapters of this report are independent, as described below.

1. **Dynamic SQL in the SAMeDL**. Dynamic SQL is a facility, not part of ANSI standard SQL but supported by many DBMS vendors, which allows a DBMS application to generate SQL statements dynamically. A dynamic SQL application can decide at runtime what DBMS interactions it needs. This can imply that the parameter profiles of these interactions are not known until runtime. This is not the way Ada likes to do things. Chapter 2 illustrates several solutions to this problem.
2. **Database Definition in the SAMeDL**. The database description facilities of the SAMeDL, particularly the view and table declarations of [18], Section 4.2, are essentially those of SQL. These facilities operate as though a database were defined once and for all. In fact, however, database definitions evolve and are modified incrementally. Furthermore, database applications occasionally have a need to execute database definition dynamically. The storage of

[1] The members of this committee are identified in the cited technical reports.

temporary results, tables that are created and destroyed during the execution of an application, is the most prevalent need. These issues are discussed in Chapter 3.

3. **Support for Multiple Concurrent Transactions in the SAMeDL**. Where Dynamic SQL presents difficulties for Ada programs, Ada's tasking features present difficulties for an SQL DBMS. A DBMS must identify each of its clients in some way and it can conceive of doing only one thing (executing only one SQL statement within exactly one transaction) for any one client at any one time. The DBMS has no insight into the task structure of an Ada program, so it has no way of identifying the tasks within an Ada program. On the other hand, an Ada software designer might wish to design a collection of tasks, each of which is meant to execute as an independent transaction. Chapter 4 describes techniques for constructing Ada applications that wish to present multiple, simultaneous transactions to a DBMS.
4. **SQL Decimal Support in the SAMeDL**. The lack of support for decimal arithmetic and decimal representations in Ada has been the subject of much discussion [1]. Chapter 5 presents a suggested implementation of decimal support. Interested readers are advised to examine the decimal support being prepared as part of the Ada9X revision project [2]. This chapter discusses the steps needed to add decimal support, in the form of base domains, to the SAMeDL. The Ada semantics for those base domains can be supplied by anyone, including the authors of [2].

The remainder of this introduction discusses issues that we considered important, but not sufficiently complicated to warrant a chapter of their own. These issues are:

- Separate and Re- Compilation of Abstract Modules
- Database Schemas and SAMeDL Schema Modules
- The Context of Derived Ada Packages
- Predefined Ada Types in SAMeDL

## 1.1. Separate and Re- Compilation of Abstract Modules

The SQL semantics paragraph of Section 5.1 (Abstract Modules) of the reference manual reads as follows:[2]

> There is an SQL module associated with each abstract module that gives the SQL semantics of the abstract module. The name of the SQL module is implementation defined. The language clause of the SQL module shall specify Ada. The module authorization clause is implementation defined.

In the SQL standard [4], Syntax rule 2 of Section 7.1 reads:

> A <module> shall be associated with an application program during its execution. An application program shall be associated with at most one <module>.

[2]In the version of the reference manual published as an SEI technical report [8], a paragraph like this appeared in Section 3.1 (Compilation Units), assigning a single SQL module to an entire SAMeDL compilation. This change simplifies the language description without changing the language.

Combined, these statements seem to imply that the SQL statements for an Ada application program must all appear in one SAMeDL abstract module. This is not the case.

SAMeDL modules are meant to be used the way packages are used in Ada. Each module forms a small, logically coherent part of one or more application programs. The database services needed by an application are provided by any number of modules, many of which may be reused by other applications. If the language quoted above is taken literally, this kind of modularization and reuse cannot occur.

The key phrase that allows us to get around this apparent difficulty lies in the sentence that introduces the Interface Semantics paragraphs of Sections 5.2 (Statements) and 5.5 (Cursor Procedures). This sentence reads:

> A call to the Ada procedure $P_{Ada}$ shall have effects that cannot be distinguished from the following.

The words "cannot be distinguished" were chosen with care. They mean that the format details of the runtime support code generated by a SAMeDL compiler are irrelevant. It matters only that that code behave *as though* it conformed to the description in the reference manual. It does not matter how the procedures are implemented, provided they have the indicated meaning. It is completely acceptable for a SAMeDL compiler to generate no module language text at all. Indeed, given the scarcity of module language compilers, it is likely.

The language *allows* separate compilation, but it does not demand it. It is up to the SAMeDL implementor to ensure that separate compilation is possible. The implementor should also take care to minimize the disruptive effects of module recompilation. If the Ada semantics of the module (i.e., the derived Ada specifications) have not changed, but the SQL semantics (e.g., the **where** clause) have, then recompilation need not affect the Ada application. This is possible only if the SAMeDL compiler implementation makes it feasible to re-compile the source text corresponding to the SQL module without re-compiling its Ada specification.

## 1.2. Database Schemas and SAMeDL Schema Modules

The SAMeDL Schema Module language, given in Section 4.2 of the reference manual, is modeled after the schema definition language in Section 6 of the SQL standard [4]. The schema definition language assumes the database is defined once and for all, an unrealistic assumption that is discussed at length in Chapter 3. The discussion here concerns the relationship between the SAMeDL database definition as described in the schema modules and the DBMS database definition.

In the SQL standard, a table has a name relative to the schema in which its declaration appears. Thus a table's full name is `schema_name.table_name`. Likewise, tables in the SAMeDL have names relative to the schema modules in which their declarations appear; so in the SAMeDL as well, a table's full name has the dotted form `schema_module_name.table_name`. But, as described in the database definition note, schema modules are not one-to-one with schemas. In fact, the SAMeDL reference manual *does not define* the database table identified by a table declaration within a schema module.

The identification of a table name in a SAMeDL schema module with a table known to the DBMS has to be made by the SAMeDL compiler implementation. The reason for this is that the connection between table names in programs and tables in the database is made in a DBMS implementation-specific way. Many DBMS have a table grouping concept like the schema, although it is often called a database. Many of those DBMS allow access to only one such group or database by any application program at a time. The identity of the database accessed is supplied at runtime. Whatever form that runtime facility takes, it is not covered by the SQL standard. In the case of IBM's DB2, the database is identified in the job control language, that is, outside of the application program and its SQL support. In many other cases, e.g., Ingres Corporation's INGRES, a user sign-on command, CONNECT, identifies the database to be accessed.

For the class of DBMS described, once the connection has been made, all table names are interpreted within the accessed database. So, at the level of the table name and column name, identification of a SAMeDL name with a DBMS name is by name equivalence. Thus the table identified by a SAMeDL table_definition with name *N* is the table with name *N* within whatever area, database, or other DBMS name space to which an application, using an abstract module referencing the schema module within which the table_definition is found, is connected. So it need not be the case that a schema module and the table declarations it contains are irrevocably and uniquely identified with one DBMS structure and some subset of tables that it contains. A given schema module may map to more than one such area, by different applications or by different invocations of the same application. This feature is actually beneficial, rather than an artifact of other decisions, when "test" versions of a production database are used. Only the name of the database, and not the application software itself, need be changed when a newly developed or modified application is put into production.

## 1.3. Optional Base Domain Options

The so-called fundamental base domain options (see Section 4.1.1.3 of [8]) are sufficient for the purposes of defining the SAMeDL. To specify precisely a value at the abstract interface defined by a SAMeDL procedure (for a fetch statement, say), it is necessary to specify precisely the values at the concrete interface *and* the process applied to those values to produce the abstract interface value. For a SAMeDL compiler to implement that specification, it will need to know more.

Consider the declaration of the *Weight* domain, as defined in the Parts Suppliers Database of Section 4.1.3 of [8].

```
domain Weight is new SQL_Int(
  First => 0,
```

```
    Last  => Max_SQL_Int);

-- Note: This text must have visibility to
-- the SAMeDL standard definition modules
-- SAMeDL_Standard and SAMeDL_System.
```

from which, using the domain pattern for SQL_Int (see Appendix C.1) the following Ada type and package declarations are derived:[3]

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

For these declarations to compile, the package specification in which they appear must have visibility to the SAME support package SQL_Int_Pkg.[4] The name "SQL_Int_Pkg" appears nowhere in SAMeDL_Standard. A SAMeDL compiler probably needs the option

```
for support package use 'SQL_Int_Pkg';
```

which it interprets as meaning "Add this package to the context of any Ada-generated code using this base domain." The compiler *could* hard-code that information for the standard base domains, rather than accept it as an option, but that would not extend to user-written base domains. A SAMeDL compiler should not special-case the standard base domains.

As another example of the need for "optional options," consider the Ada code that must be generated to move a weight from the concrete to the abstract interface, assuming null values are possible. Here is an informal (and incorrect) version of such code:

```
if indicator_variable >= 0 then
  target_variable := Weight_Ops.With_Null(source_variable);
else
  target_variable := ???????
end if;
```

There are two problems. First, Weight_Type, the null-bearing type of *target_variable*, is limited, so predefined assignment is not available. The compiler will need to know that and the identity of the assign procedure to use. One possibility is an option of the form

```
for null-bearing assign use '[self]_Ops.Assign';
```

added to the specification of the base domain SQL_Int. Since the not null type is visible, the option

```
for not-null-bearing assign use predefined;
```

[3]The upper bound on the range of Weight_Not_Null is the largest positive integer representable by the DBMS, which is recorded in the constant Max_SQL_Int in the definition module SAMeDL_System.

[4]Or some other package exporting the types SQL_Int and SQL_Int_Not_Null and the package SQL_Int_Ops.

states that Ada predefined assignment (:=) may be used.

Second, the compiler needs access to a null value of the correct type, to fill in for the question marks above. The option

```
for null value use 'Null_SQL_Int';
```

supplies this information. With this information, it is possible to generate a correct version of this code:

```
if indicator_variable >= 0 then
  Weight_Ops.Assign(target_variable,Weight_Ops.With_Null(source_variable));
else
  Weight_Ops.Assign(target_variable,Null_SQL_Int);
end if;
```

This code will work *provided* that it has **use** visibility to the Ada package derived from the SAMeDL definition module within which the Weight domain is defined.

## 1.4. Previously Defined Ada Types in SAMeDL

The SAMeDL tacitly assumes a model of database application development in which database definition is independent of and prior to application development. Therefore, the Ada types describing database data are part of the database definition and of the SAMeDL supplied services, not part of the application. However, when a DBMS is added to an existing application suite, it may be preferable for the database data to have types declared in the application. The SAMeDL base domain capability allows this.

The simplest way to accomplish the re-use of an Ada type is to define a base domain specifically for that purpose. Assume that the type to be re-used is *T* and that it is declared in the specification of the package *P*. A set of SAMeDL definitions that will allow *T* to be the type of SAMeDL parameters is given by:

```
definition module Reuse_T is
  base domain Reuse_T_BD is      -- no parameters needed
    domain pattern is
      'subtype Reused_T is P.T;'
    end pattern;

  for not null type name use 'Reuse_T';
  for null type name use 'Reuse_T';
  for data class use data_class;   -- as needed
  for dbms type use dbms_type [pattern_list];   -- as needed
  for conversion from type to type use converter;   -- as needed
  for support package use 'P';

  end Reuse_T_BD;

  domain T_Reused is new Reuse_T_BD not null;
end Reuse_T;
```

from which the following Ada code is derived

```
with P;
package Reuse_T is
  subtype Reused_T is P.T;
end Reuse_T;
```

Note that the support package option introduced earlier creates the appropriate context (**with**) for this Ada package. It is easy enough to generalize this base domain to one that can be used for any type exported in a package specification by parameterizing the type and package name as follows:

```
base domain Reuse_Any_Type
  (Type_Name : character;
   Package_Name : character)
  domain pattern is
    'subtype Reused_[Type_Name] is [Package_Name].[Type_Name];'
  end pattern;

  for not null type name use 'Reuse_[Type_Name]';
  for null type name use 'Reuse_[Type_Name]';
  for data class use data_class;  -- as needed
  for dbms type use dbms_type [pattern_list];  -- as needed
  for conversion from type to type use converter;  -- as needed
  for support package use [Package_Name];

end Reuse_Any_Type;
```

and now the following definition module will generate the same Ada package specification as given earlier.

```
definition module Reuse_T is
  domain T_Reused is new Reuse_Any_Type not null
    (Type_Name => 'T',
     Package_Name => 'P');
end Reuse_T;
```

These examples have assumed that there is no need to support null values when storing objects of a pre-existing Ada type in an SQL database. The generic package in Figure 1-1 will generate a null-bearing type and conversion functions that effectively add null value support to any existing type. A base domain that can be used to declare domains supporting null values and based on pre-existing Ada types is given by:[5]

```
base domain Add_Null_to_Any_Type
    (Type_Name : character;
     Package_Name : character)
  domain pattern is
      'subtype [self]_Not_Null is [Package_Name].[Type_Name];'
      'package [self]_Ops  is new Extend_With_Null'
        '(Type_to_Reuse => [Package_Name].[Type_Name]);'
```

[5]This base domain uses naming conventions more like those of the standard base domains, but this is purely conventional.

```
        'type [self]_Type is new [self]_Ops.Type_with_Null;
    end pattern;

    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use data_class;   -- as needed
    for dbms type use dbms_type [pattern_list];   -- as needed
    for conversion from not null to null use function With_Null;
    for conversion from null to not null use function Without_Null;
        -- dbms conversions as needed

    for support package use [Package_Name];
    for support package use Extend_With_Null;

    for null value use 'Null_Value';
    for null-bearing assign use 'Assign';

  end Add_Null_to_Any_Type;
```

Notice that this base domain needs two support packages. Finally, an example of the use of this base domain and the derived Ada code follows:

```
  definition module Reuse_with_Null is
    domain Reuse_T is new Add_Null_to_Any_Type
      (Type_Name => T, Package_Name => P);
  end Reuse_with_Null;

  with P, Extend_with_Null;
  package Reuse_with_Null is
    subtype Reuse_T_Not_Null is P.T;
    package Reuse_T_Ops is new Extend_With_Null
      (Type_to_Reuse => P.T);
    type Reuse_T_Type is new Reuse_T_Ops.Type_with_Null;
  end Reuse_with_Null;
```

```
generic
    type Type_to_Reuse is private;
package Extend_With_Null is
    type Type_with_Null is limited private;
    function With_Null (Left : Type_to_Reuse) return Type_With_Null;
    function Without_Null (Left : Type_with_Null) return Type_to_Reuse;
    function Is_Null (Left : Type_with_Null) return boolean;
    procedure Assign (Left : out Type_with_Null;
                      Right : Type_with_Null);
    function Null_Value return Type_With_Null;

    private
        type Type_With_Null is record
            Is_Null : boolean := true;
            Value : Type_to_Reuse;
        end record;
end Extend_With_Null;

with SQL_Exceptions;
package body Extend_With_Null is
    function With_Null (Left : Type_to_Reuse)
        return Type_With_Null is
    begin
        return (False, Left);
    end With_Null;
    function Without_Null (Left : Type_with_Null)
        return Type_to_Reuse is
    begin
        if Is_Null(Left) then
            raise SQL_Exceptions.Null_Value_Error;
        else
            return Left.Value;
        end if;
    end Without_Null;
    function Is_Null (Left : Type_with_Null)
        return boolean is
    begin
        return Left.Is_Null;
    end Is_Null;

    function Null_Value return Type_With_Null is
        Null_Hldr : Type_with_Null;
    begin
        return Null_Hldr;
    end Null_value;
    procedure Assign (Left : out Type_with_Null;
                      Right : Type_with_Null) is
    begin
        Left.Is_Null := Right.Is_Null;
        Left.Value := Right.Value;
    end Assign;
end Extend_with_Null;
```

**Figure 1-1:** Support Package for Extending with Nulls

# 2. Dynamic SQL in the SAMeDL

**by Gary Chastek and Marc H. Graham**

## 2.1. Introduction

The SAMeDL (see [8]) is designed to facilitate the construction of Ada application programs that access and manipulate data stored in a database management system whose data manipulation language is ANSI standard SQL (see [4]). Dynamic SQL is frequently offered by commercial database management systems (see [14] and [12]), but is not part of the standard. Hence the SAMeDL does not currently support dynamic SQL. The SAMeDL can, however, be extended to support dynamic SQL. This chapter discusses how the SAMeDL may be extended to provide such support.

Dynamic SQL presents a problem to any Ada program. Ada does binding and type checking statically: name resolution occurs at compile time. Dynamic SQL, however, binds dynamically; names, even the statements to be executed, are not necessarily known until runtime.

Dynamic SQL is currently offered by various vendors using various dialects. It is inappropriate for our purposes to select a particular vendor's dialect of dynamic SQL, and beyond the scope of this work to consider a large subset of vendors that offer dynamic SQL. Therefore, the follow-on standard SQL2 was chosen as our SQL model [16]. SQL2 is still being developed;[6] therefore, the statements we make about it may not be true of the final standard. Nonetheless, it serves as a good example of what can be expected from dynamic SQL facilities.

The next section presents a brief description of dynamic SQL as it appears in SQL2, and the following section illustrates various potential implementations of support for dynamic SQL by a SAMeDL compiler. Included in these illustrations is a complete example, in SAMeDL and Ada, of a procedural interface to dynamic SQL.

[6]Recently, the proposal of [16] has been accepted by ISO as a Draft International Standard (DIS), the final step before being accepted as an International Standard. The DIS is available as [10]. The primary difference between [16] and [10] is the section numbering. The dynamic SQL section, which was Section 12 of [16], is Section 17 of [10].

## 2.2. Dynamic SQL in SQL2

### 2.2.1. Introduction

SQL, as defined by the ANSI standard ( [4]), is static; that is, the names and types of the parameters are known at compile time. Parameters may represent objects whose values, for example, appear in search conditions or as values to insert into a table. Most SQL statements can take parameters. These parameters, however, can only appear in those parts of an SQL statement in which a constant may appear. Further, the number and data types of these parameters must be known at compile time.

As an example of a situation for which static SQL is insufficient, suppose a SAMeDL processor needs to determine whether a given identifier is the name of a column in exactly one table in a list of tables (the "current scope"). The program might maintain a data dictionary containing a table *Columns* with columns *Table_Name*, *Column_Name* (plus others), so that *C* names a column in Table *T*, precisely when the pair *TC* appears in the *columns* table. Now, if the *number of tables in scope* is known, it is possible to write a SAMeDL procedure containing an SQL `<select statement>` that will retrieve just this information. If there are three tables in scope, then the procedure is:[7]

```
procedure Dictionary_Lookup
    (Column_Name_In : Column_Name;
     Table1         : Table_Name;
     Table2         : Table_Name;
     Table3         : Table_Name)
is
    select Table_Name
    from Columns
    where Column_Name_In = Column_Name and
          (Table_Name = Table1 or
           Table_Name = Table2 or
           Table_Name = Table3)
    status Just_One;
```

But, of course, the number of tables in scope is not constant; it varies from statement to statement. No single static SQL statement will solve this problem for all possible scopes. Using dynamic SQL, the processor can create a different statement for each scope.

The next two sections describe dynamic SQL in SQL2. The first section presents an overview of how dynamically specified statements are handled in SQL2, while the second section discusses how parameters are passed to and from dynamic SQL statements.

[7]The status map `Just_one`, which is not shown as it is not particular to dynamic SQL, returns a status parameter value which distinguishes among the cases: zero, one, or multiple answers. The processor can use that information to distinguish a correct usage from the two possible incorrect usages.

### 2.2.2. Dynamic SQL Statements

An SQL statement that is to be dynamically executed is created by the application program as a string, and presented to the DBMS as the operand of a *PREPARE* statement. For example, if STMT_TO_PREP is a character string and ST an identifier, then the statement

```
PREPARE ST FROM STMT_TO_PREP;
```

will verify that the string that is the value of the parameter[8] STMT_TO_PREP represents a valid SQL statement. If no errors are found, the statement is encoded, and may be referred to in later dynamic SQL statements by referencing the identifier ST.

If the prepared SQL statement is not a *SELECT* statement[9] and requires no parameters (parameters will be discussed in the following section), the prepared statement may then be executed:

```
EXECUTE ST;
```

If, in this case, the statement (again, not a *SELECT* statement) contained in STMT_TO_PREP is to be executed only once, then the *PREPARE* and *EXECUTE* could be done in one step, using an *EXECUTE IMMEDIATE*:

```
EXECUTE IMMEDIATE STMT_TO_PREP;
```

If the prepared statement is a *SELECT* statement, then a cursor must be declared for the *SELECT* statement. Once declared, the cursor is dynamically opened, fetched, and closed in much the same manner as in static SQL. For example:

```
DECLARE MY_CURSOR CURSOR FOR ST;
OPEN MY_CURSOR;
FETCH FROM MY_CURSOR;
CLOSE MY_CURSOR;
```

It is important to note the distinction between dynamic SQL statements and SQL statements that are dynamically executed. SQL2 support for dynamic SQL is provided by dynamic SQL statements, such as *PREPARE*, *EXECUTE*, *EXECUTE IMMEDIATE*, etc.; these statements are used to process the other, non-dynamic SQL statements, such as *SELECT*, *UPDATE*, and *INSERT*. The dynamic SQL statements are, themselves, static, i.e., known at compile time. The operands to the dynamic SQL statements, such as the SQL statements to prepare or execute, are not, however, known until runtime.

---

[8]Assume that the statements in this section appear in an SQL module. STMT_TO_PREP is a parameter to the encompassing module procedure.

[9]By the time this is printed, these restrictions on SELECT statements may have been lifted, effectively allowing non-cursor, single-row select statements to be dynamically executed. As mentioned earlier, these uncertainties in SQL2 are unimportant for the purposes of this chapter.

### 2.2.3. Dynamic SQL Parameters

The previous section discussed dynamic SQL statements that did not involve parameters. Parameters are associated with dynamic SQL statements by the `<using clause>`. The operand of the SQL2 `<using clause>` (see Section 12.1 in [16]) specifies how dynamic SQL runtime parameters are described. There are two cases: parameters may be described by a list of identifiers, or by an SQL descriptor.

In the simpler case, the operand of the `<using clause>` is a list of identifiers. In this case, the number, order, and types of parameters are set at compile time; the dynamic part is elsewhere in the statement (e.g., in the `<where clause>`).

Consider again the problem of looking up column names in the data dictionary. For each scope (i.e., list of table names), the SQL query that will look up column names in that scope is given by

```
SELECT Table_Name
FROM Columns
WHERE Column_Name = ? AND
      (Table_Name = T1 OR
       Table_Name = T2 OR
       .  .  .
       Table_Name = Tn);
```

where $T_1$,...$T_n$ is the list of table names in scope. The question mark in the equality comparison `Column_Name = ?` is a so-called *dynamic parameter* ( [16], Section 5.7) and occupies the location of an input parameter; the statement can be used to look up any putative column name. Notice that the variability in this statement is in the list of table names, which are coded as literals, as they are known at the time the statement is created.

As described above, this statement must be *PREPARE*d and then *DECLARE*d to be the query specification of a cursor.[10] That cursor (`MY_CURSOR`, as above) is opened with the statement

```
OPEN MY_CURSOR USING NAME_TO_FIND;
```

where NAME_TO_FIND contains the identifier being looked up. The rows, if any, defined by the query are accessed by the statement

```
FETCH MY_CURSOR USING TABLE_FOUND;
```

If the number and types of the parameters to a statement are *not known* at the time at which the program is written, then the parameter values will be passed via an SQL *descriptor area* (i.e., the `<using clause>` takes the form `USING descriptor name`). The SQL descriptor area is a table that describes the input or output parameters of a dynamically executable statement that has been prepared. Each parameter has an entry in the table that describes the parameter's type, length, precision, scale, and value.

[10]This assumes that dynamic SQL does not support the single row select.

The SQL descriptor area is treated like an Ada limited private type; an application cannot directly manipulate an SQL descriptor area, but rather must use the area's associated SQL descriptor name and the supplied operations to access and manipulate the SQL descriptor area's fields.

The typical use of an SQL descriptor starts with an *ALLOCATE* to allocate space for, and associate a unique name with, a descriptor area. If input parameters are to be described using a descriptor, then one must be allocated:

```
ALLOCATE DESCRIPTOR IN_DESCRIPTOR;
```

The initial values associated with the descriptor are undefined.

The statement to be executed is then named and encoded by a *PREPARE* statement in the same manner as described in the previous section.

```
PREPARE ST FROM STMT_TO_PREP;
```

The *DESCRIBE* statement can then be used to extract the parameter information from a prepared statement into a descriptor area. The following *DESCRIBE* statement will be needed for the input parameters.

```
DESCRIBE INPUT  ST USING IN_DESCRIPTOR;
```

Information about the parameters to the prepared statement, such as the number, types, and values of the parameters, can then be obtained using the *GET DESCRIPTOR* statement. This statement can be used to determine, for the specified descriptor (1) the number of parameters associated with that descriptor, or (2) the attributes (type, length, value, etc.) of a particular parameter associated with that descriptor.

The following example shows both uses. In the first *GET DESCRIPTOR* statement, the keyword COUNT represents the number of associated parameters. In the second statement, the keyword VALUE indicates that an attribute of the parameter, whose position is indicated by the number following VALUE, in this case 1, is to be returned; the keyword TYPE specifies the type code of parameter number 1. The example assumes the existence of two exact numeric parameters with scale 0, NUM_PARAMS, and PARAM1_TYPE, which will contain, respectively, the number of parameters in the input parameter list for ST and the type of the first parameter in that list.[11]

```
GET DESCRIPTOR IN_DESCRIPTOR NUM_PARAMS = COUNT;
GET DESCRIPTOR IN_DESCRIPTOR VALUE 1 PARAM1_TYPE = TYPE;
```

If the descriptor is to be used as an input parameter list, then a *SET* statement is used to specify the values of the input parameters. If the previous *GET DESCRIPTOR* statement indicated that the first parameter was an integer, then that parameter's value (i.e., its DATA field in the descriptor area) could be set to 12 by:

[11]Actually, PARAM1_TYPE will contain a code for the parameter's type, as specified in [16], Clause 12.1.

```
SET DESCRIPTOR IN_DESCRIPTOR VALUE 1 DATA = 12;
```

Note that the *SET DESCRIPTOR* statement's use is similar to the use of the *GET DESCRIPTOR*, except that in the *SET DESCRIPTOR* statement, the keyword indicating the desired attribute appears to the left of the equals sign.

If the SQL statement to execute will have associated output parameters that are to be described by a descriptor, then another descriptor must be allocated:

```
ALLOCATE DESCRIPTOR OUT_DESCRIPTOR;
```

The desired SQL statement is then executed:

```
EXECUTE ST INTO OUT_DESCRIPTOR
           USING IN_DESCRIPTOR;
```

The "INTO OUT_DESCRIPTOR" in the *EXECUTE* statement specifies that the descriptor named "OUT_DESCRIPTOR" describes the output parameters for ST's execution. The "USING IN_DESCRIPTOR" performs a similar function for the input parameters to ST.

Finally, the *GET DESCRIPTOR* statement is again used, this time to extract the values from the output parameters.

```
GET DESCRIPTOR OUT_DESCRIPTOR NUM_PARAMS = COUNT;
GET DESCRIPTOR OUT_DESCRIPTOR VALUE 1 PARAM1_TYPE = TYPE;
```

If the previous *GET DESCRIPTOR* statement indicated that the first parameter was an integer, that parameter's value (i.e., its DATA field in the descriptor area) could then be retrieved by:

```
GET DESCRIPTOR OUT_DESCRIPTOR VALUE 1 PARAM1_DATA = DATA;
```

where PARAM1_DATA is assumed to be of an integer data type.

## 2.3. SAMeDL Support for Dynamic SQL

### 2.3.1. Introduction

Dynamic SQL presents a serious challenge to any Ada SQL interface mechanism that, like the SAMeDL, extends Ada-like type checking to the interactions of an Ada program with a database. To get a grasp of the difficulties involved, consider the following list, taken from Section 1.5 of [9]), of "requirements for the SAMeDL":

1. Modular program construction; separate compilation.
2. Application oriented, strong typing

    - within the description of the database interaction (i.e., within the SAMeDL); <u>and</u>
    - at the application interface.

3. A safe treatment of missing information (null values); safe in the sense that missing information cannot be mistaken for real information.
4. Robust status parameter handling.

The procedures of a SAMeDL module can contain dynamic statements (DESCRIBE, PREPARE, EXECUTE, etc.), but cannot, by definition, contain dynamically executed statements, as these do not exist at the time the SAMeDL module is written. Since the dynamically executed SQL statement is a string object within the application, it cannot be said that the application is isolated from the details of the database and of SQL; indeed, quite the contrary.[12] Therefore much of the benefit of SAMeDL modularization does not accrue to dynamic SQL applications. On the other hand, and at the other end of the list above, SQLCODE values for dynamic statements include those for the dynamically executed statements, plus new error conditions for the dynamic statements. In any case, the SAMeDL status clause may be attached to dynamic statements in the regular way. SAMeDL robust status parameter handling carries over to the dynamic case without modification or diminution.

### 2.3.2. Strong Abstract Typing and Dynamic SQL

Of the remaining issues, the most problematic is the application of SAMeDL type checking to the dynamically executed statements. There are a number of ways in which this problem can be addressed. The minimal approach is to do nothing at all; SAMeDL type checking is not applied to dynamically executed statements. This is, obviously, easy to implement, and it is the method we suggest in the remainder of this note. Some ways in which SAMeDL checking might be applied are presented below.

The most obvious technique is to include a SAMeDL parser and type checker in the runtime system. The abstract procedure for a PREPARE statement would then parse the statement by the SAMeDL rules, which may require, *inter alia*, access to the SAMeDL data dictionary. Notice that the SQL procedure for a PREPARE statement will parse by the rules of SQL, which also requires data dictionary access. Under optimal conditions, the two parses could be done simultaneously.

It's possible to build a dynamic SQL generation facility. This is a collection of Ada subprograms, appropriately packaged, that can be used to create an SQL statement guaranteed to be syntactically acceptable by the rules of SAMeDL. As an example of the functionality such a facility might offer, consider a procedure to create select statements, the parameter profile of which might be:

```
function Create_Select (
          Select_List          : Select_List_Type;
          Table_List           : Table_List_Type;
          Where_Clause         : Where_Clause_Type;
```

[12] Of course, this does not prevent that part of the application that performs the dynamic SQL interaction from being isolated from that part of the application that uses the logical data service. But the isolation is not provided by the SAMeDL module.

```
        Group_By              : Group_By_Type;
        Having                : Having_Type;
        Order_By              : Order_By_Type)
    return Statement_Type;
```

Other functions create select lists, where clauses, and so forth. The Create_Select function body can verify, at runtime, that the value expressions in the select list are well formed, the table list contains tables actually defined, etc. This still will need data dictionary access at runtime, but it will save the effort of lexical analysis and parsing at runtime, although that effort will be expended by the SQL engine in any case. Such a generation facility is likely to be easier to use and less error prone than a simple string facility. The so-called WIS approach [7] to SQL Ada interfacing is an example of a dynamic SQL generation facility.

There is an alternative that is particularly appealing to applications that are only "slightly dynamic." The processor column lookup problem introduced earlier is an example of such an application. Recall that the variability in the statement of that problem is confined to the list of table names. The rest of the statement is known at the time the program is written and could be submitted to the SAMeDL processor. Consider the following text, which has the form of an `extended_procedure`.

```
extended procedure dynamic statement Column_Search
    (Column_Name_In : Column_Name)
is
    SELECT Table_Name
    FROM Columns
    WHERE Column_Name = Column_Name_In AND <predicate> ;
```

The token `<predicate>` is a non-terminal of the grammar of the SAMeDL. Its appearance represents a parameter to the PREPARE for the statement Column_Search. This PREPARE procedure, which might be automatically generated, takes a string parameter whose value must parse as a SAMeDL predicate and that will replace the token `<predicate>` when the statement is prepared. In other words, there is an abstract procedure

```
extended procedure Prepare_Column_Search
    (Predicate : Predicate_Type)
is
    prepare Column_Search;
```

that could be called by a statement like

```
Prepare_Column_Search(
  Predicate =>
    "(Table_Name = T1 OR Table_Name = T2 OR Table_Name = T3)");
```

Roughly, this alternative suggests an extended quasi-procedure called a dynamic statement, a SAMeDL statement with embedded non-terminal tokens (`<predicate>`, `<column_reference>`, etc.). The processor can do full, compile-time type checking on the fully specified parts of the statement and check that the non-terminal tokens are used appropriately. The run time system can be of either the parse or the generate variety.

This is the only alternative that allows for the automatic generation of correctly and abstractly typed procedure declarations of the abstract interface procedures called by the application program. This is also the only alternative that may have enough information *at compile time* to do that generation. For the other alternatives, the information does not become available until runtime.

This is the most difficult alternative to implement, as it presupposes one of the other run time implementations. A parser for dynamic statements as defined here is roughly identical, but slightly larger, than a parser for SAMeDL statements. An implementation of this alternative for dynamic statements will, by definition, have a SAMeDL parser available to be copied. Therefore, this treatment of dynamic statements is a considerable, but not a heroic, undertaking.

### 2.3.3. A Procedural Dynamic SQL Implementation

We now present a simple, procedural dynamic SQL implementation. It is simple in that it does not type check the dynamically executed SQL statement, although it could be coupled with either the parse or the generate alternatives described above. It is procedural in that it consists of a fixed set of procedure declarations that can be invoked by any dynamic SQL application. Such a procedural interface is possible for dynamic SQL since the linguistic variation, the SQL statement, appears as an actual parameter in a dynamic SQL setting.

A suggested procedural interface for dynamic SQL, given as a collection of extended procedures in an extended abstract module, is shown in Figure A-1. Essentially, each dynamic SQL statement in Sections 12.2 through 12.15 of [16] is turned into a procedure in a straightforward way. The domains used to declare the parameters in those procedures are defined in the definition module Dynamic_Domains in Figure A-2. The domains Descriptor_Name, Dynamic_Statement, Dynamic_Statement_Identifier, and Dynamic_Cursor_Identifier are character strings with implementation-defined length. The value of an object of the Dynamic_Statement domain is an SQL statement; the value of an object of the other of these domains is a name (or identifier) of a descriptor, statement, or cursor.

The domains Dynamic_Char, _Int, _Smallint, _Real and _Double_Precision are the domains used for parameters whose application domain is not known when the application is written. For domains other than Dynamic_Char, these are just unconstrained domains based on the SAMeDL standard base domains.

Dynamic_Char needs a new base domain that can accommodate strings of any length. This base domain definition, SQL_Unconstrained_Char, is found in the definition module Dynamic_Base_Domains in Figure A-3. Notice that its pattern is identical to SQL_Char with the constrained subtype declarations removed from its patterns. The SQL2 proposal [16] has a varying length character string data type, VARCHAR. SQL_Unconstrained_Char may serve as a model for the Ada type declarations to support VARCHAR. The **dbms type** option of SQL_Unconstrained_Char specifies VARCHAR, an implementation-defined dbms type, assuming an implementation for such a type exists in the target DBMS.

Dynamic_Base_Domains also contain the base domain Dynamic_Parameter_Base, which exists for the sole purpose of declaring the domain Dynamic_Parameter, used in the procedures Get_ and Set_Parameter. The Ada type Dynamic_Parameter is a variant record that may hold the value of any dynamic parameter. Notice that only null bearing types are supported for dynamic parameters, simplicity, and generality. A type Dynamic_Parameter_Not_Null, for holding any non-null value of a dynamic parameter, is conceivable.

The Ada package Dynamic_Domains, derived from the definition module of that name, appears in Figure A-4. The Ada package Dynamic_Stmts, derived from the abstract module Dynamic_Stmts, appears in Figure A-5.[13]

The procedures in Dynamic_Stmts use the descriptor form of parameter passing, as it is more general. The procedures Get_Parameter and Set_Parameter transfer all the information about a parameter to or from an Ada application in a single call.[14] Alternatively, these procedures could have been coded with a fourth parameter to indicate which information to get or set.

A portion of the body of Get_Parameter is shown in Figure A-6. The package Concrete_Dynamic_Stmts contains the Ada declarations of the concrete SQL module containing SQL dynamic statements and is not shown. The Get_Value procedure assumes some implementation of VARCHAR to support the retrieval of strings of any length in a module language procedure.

### 2.3.4. Problems with the Procedural Interface

The procedural interface illustrated by the module Dynamic_Stmts has some weaknesses that may or may not be significant to a given application. First, it requires the so-called extended form of descriptor, statement, and cursor identifiers; it does not allow them to be coded as literals in the dynamic statement. This requires the application programmer to keep track of these things. Further, a target DBMS may not support these extended forms. The SQL2 defined "Intermediate Level" SQL does not have extended descriptor names.

The dynamic cursor update and delete statements ( [16], Clauses 12.16 and 12.17) cannot be supported by a procedural interface. The `<table name>` must be coded into these statements. Therefore they must be prepared especially for each application that needs them.

---

[13] The SAMeDL modules Dynamic_Stmts, Dynamic_Domains, and Dynamic_Base_Domains are not strictly needed for a procedural interface to dynamic SQL; the Ada packages Dynamic_Stmts and Dynamic_Domains are sufficient. Some SAMeDL support is needed for the dynamic cursor update and delete statements, so SAMeDL modules like these are likely to be needed, eventually.

[14] These procedures do not support the NULLABLE attribute.

Finally, the procedural interface can use only the descriptor form of parameter passing. This form imposes a requirement of generality on the application program that is often unnecessary. If the parameter profile of a dynamic statement is known when the program is written, the identifier list form of the `<using clause>` is preferred.

Consider once again the example of data dictionary lookup presented earlier. This statement always takes a column name as input and returns zero or more table names. The application would prefer not to have to deal with the generality of Dynamic_Parameter's and thus prefers an identifier list form of the `<using clause>`. It can still use the form of the PREPARE statement given in the Dynamic_Stmts module, but it also uses the statements in the module Example_Using of Figure 2-1.

The Ada declarations for Example_Using are not shown, but are derived in a straightforward way. In particular, no subpackage corresponding to the cursor is generated in the Ada package. A record type might be generated for the FETCH, and an `into_clause`( [8], Section 5.9) might be used to control it.

Unlike the case of a static cursor in the SAMeDL, the example in Figure 2-1 does not treat the cursor as an entity. Such a treatment is possible. A dynamic cursor might be defined by the following grammar rule:

```
extended dynamic cursor cursor_name
    [with inputs ( input_parameter_list )]
    with select list dynamic_select_list ;
```

A SAMeDL `input_parameter_list` ( [8] 5.6) specifies the names and types of the input parameters to the cursor and may be used unmodified to describe the inputs of dynamic cursors. A SAMeDL `select_list` ( [8] Section 5.7) must be modified slightly to describe the outputs of dynamic cursors. The `value_expressions` of a SAMeDL select parameter cannot be used here. Instead, a `dynamic_select_list` might use domain names, as the domain of the select parameter is all that is of use here. Thus the Example_Cursor in Figure 2-1 could be coded by

```
extended dynamic cursor Example_Cursor
        with inputs (Column_Name_In : Column_Name not null)
        with select list Table_Name named Table_Found;
```

The procedures of Figure 2-1 can be generated from this declaration. From a specification like this the SAMeDL processor cannot guarantee that the statement used as the query specification for the declaration of this cursor actually has this collection of inputs and outputs. That issue was discussed in Section 2.3.2.

```
with Dynamic_Domains, Dictionary_Domains;
abstract module Example_Using is

    extended procedure Declare_Example_Cursor
        (Cursor_Declaration : Dynamic_Domains.Dynamic_Statement)
    is
        DECLARE Example_Cursor CURSOR FOR Cursor_Declaration;

    extended procedure Open_Example_Cursor
        (Column_Name_In : Column_Name not null)
    is
    OPEN Example_Cursor USING Column_Name_In;

    extended procedure Fetch_Example_Cursor
        (Table_Found : Table_Name)
    is
        FETCH Example_Cursor USING Table_Found
        status Standard_Map;

    extended procedure Close_Example_Cursor
    is
        CLOSE Example_Cursor;
end Example_Using;
```

**Figure 2-1:** Example of Procedures with Known Parameter Profiles

# 3. Database Definition in the SAMeDL

**By Gregory Zelesnik**

## 3.1. Introduction

Database definition is a twofold process. It consists of *initial database declaration*, which is the creation of the original set of base tables in the DBMS, and *database evolution*, which is the incremental change to this initial definition that occurs over time. American National Standards Institute (ANSI) standard SQL indirectly addresses the issue of initial database declaration (see Section 3.2.1), yet does not address issues of database evolution.

Support for database definition, however, is a practical necessity. Commercial DBMS implementations provide facilities that perform the tasks of initial database declaration and database evolution. In commercial DBMS, database definition can be accomplished by processing texts of data definition language (DDL) statements with batch or interactive system software utilities, and by executing DDL from within application programs.

DDL embedded in application programs is called executable DDL. The application programmer can use it, for example, to create and delete temporary tables in the database. Temporary tables are useful for solving problems such as the elimination of common subexpressions[15] in an application.

Executable DDL and DDL statements that perform database evolution are features of commercial DBMS that are not addressed by ANSI standard SQL. Because the SAMeDL is based on standard SQL, it remains similarly silent on these issues of database definition. However, the SAMeDL was developed to facilitate the production of Ada applications that interact with commercial DBMS. To fully support the DDL of a particular commercial SQL implementation, the SAMeDL compiler implementor must be prepared to bridge the gap between database definition in ANSI standard SQL and commercial DBMS.

The next section describes database definition in ANSI standard SQL, the SAMeDL, and commercial SQL implementations. These descriptions and the surrounding discussions are intended to provide a rationale for the SAMeDL model of database definition, and compare and contrast its database definition semantics with those of commercial SQL implementations. The following section discusses the implementation of SAMeDL extensions to support commercial DDL capabilities not addressed in the SAMeDL language definition. The last section discusses the issue of keeping the SAMeDL and DBMS data dictionaries consistent.

[15] If the result of a common subexpression (an expression appearing in more than one query) is not a large relation, and can be read from the database faster than it can be computed, then it is advantageous to compute the common subexpression once, and store it in a temporary table (see Chapter 8 of [20]).

## 3.2. Database Definition

Each section below describes the semantics of database definition for one of the following: ANSI standard SQL, the SAMeDL, and commercial SQL implementations. In each section the applicable concepts and terms are defined, and the database definition semantics are discussed in terms of initial database declaration and database evolution. Each section describes executable DDL semantics as well.

### 3.2.1. ANSI Standard SQL

ANSI standard SQL only partially addresses issues of database definition. This section discusses the concepts and semantics of database definition as specified in the standard.

#### 3.2.1.1. Terminology

The primary concept of database definition in ANSI standard SQL is the *schema* (see Section 4.6 of [4]). The schema is a persistent object in the environment and is comprised of all table, view, and privilege definitions known to the environment for a specified authorization identifier. The tables, views, and privileges are *defined* by the schema definition, and cannot exist in the environment except within a schema.

The *environment* in the ANSI SQL standard is implementation defined. However, the ANSI standard implies that the environment *contains* the schema object, and that it may contain more than one (see Section 4.7 of [4]).

The *database* in ANSI standard SQL is defined as the collection of all data defined by the schemas in an environment.

#### 3.2.1.2. Database Declaration and Evolution

In [4] the semantics of database definition in ANSI standard SQL is described by:

> NOTE: An implementation may provide facilities (such as DROP TABLE, DROP VIEW, ALTER TABLE, and REVOKE) that allow the definitions of the tables, views, and privileges for a given <authorization identifier> to be created, destroyed, and modified incrementally over time. This standard, however, only addresses the <schema>s that represent the definitions known to the system at a given time.

The first sentence of the above quotation suggests that the ANSI standard recognizes a potential need for DDL that performs database evolution. The second sentence, however, implies that such DDL is not addressed by the standard. Because it is not addressed by the standard, DDL that performs database evolution must be considered nonstandard SQL.

The second sentence describes what little semantics exist for database definition in ANSI standard SQL. First, all database definition occurs through the definition of schemas. Second, the schema definition language defined by the standard is sufficient to describe the contents of the database at any given point in time.

The standard remains silent about how the schemas and the definitions they contain actually come into existence. Once they exist, however, the schema definition language of ANSI standard SQL is sufficient to describe these definitions. The standard also remains silent on issues of deleting and replacing schemas that already exist. Any DDL that deletes schemas from the environment must be considered nonstandard SQL, because such DDL is not defined by the ANSI standard. However, the same functionality may be provided by DBA tools supplied by the DBMS vendor without affecting the language. The standard omits any discussion involving the replacement of schemas as well. This leaves issues such as database evolution through schema replacement open to interpretation by the commercial DBMS vendors.

In summary, the semantics defined for database definition by the ANSI SQL standard are extremely limiting for commercial implementations. The standard does not define a means for evolving a database definition, and inadequately addresses practical aspects of initial database declaration through schema definition.

#### 3.2.1.3. Executable DDL

ANSI standard SQL does not define executable DDL. The syntax for an embedded SQL statement (see Annex A of [4]) does not include schema definition language elements (see Chapter 6 of [4]). Therefore, the standard does not address database definition from within application programs.

### 3.2.2. Commercial Implementations

Because ANSI standard SQL does not adequately address the pragmatic aspects of database definition, commercial SQL implementations typically deviate from the standard to sufficiently address these issues, especially with respect to database evolution. This section discusses database definition semantics of typical commercial implementations, and uses specific examples from existing implementations[16] to help define and illustrate the concepts.

#### 3.2.2.1. Terminology

The primary concept of database definition in commercial SQL implementations is the *database*. Although there are as many definitions of *database* as there are commercial implementations, the term is generally defined as a collection of database objects[17] and storage spaces used to store application data. In Chapter 1 of [13], DB2 defines a database as a collection of tables and associated indexes, as well as the table spaces in which they reside, used to store application data. In [19], ORACLE defines a database as a disk

---

[16]Throughout this section **INGRES** refers to INGRES version 6.2 for Unix (see [15] and [14]), **ORACLE** refers to ORACLE version 5.1A for MS-DOS (see [19] and [17]), and **DB2** refers to IBM DATABASE2 Release 2 for TSO (see [13] and [12]). The discussion of database definition semantics in this section is synthesized from the sources cited in the previous sentence. No guarantee is implied or should be inferred that the database definition semantics have been accurately interpreted.

[17]A database object is a persistent object in the database that has a data dictionary description. Tables and views are two examples of database objects.

storage area in which tables, views, and other objects are stored; ORACLE also defines it as the set of objects stored in that area. In INGRES (see [15]), the database is used to store database objects such as tables, views, indexes, and procedures.

The meaning of the term *database* defined by commercial implementations differs from its definition in the ANSI standard. In the standard database refers to data rather than a collection of database objects such as tables and views. The database in commercial implementations is more closely related to the schema object of ANSI standard SQL.

A more general concept of database definition used in commercial implementations is that of the *system*. This term is typically used instead of the term *environment*, and it refers to all database objects and data managed by a particular DBMS software installation. The system contains both application data and internal data maintained by the DBMS that describes all of the definitions in the system. The application data are maintained in one or more databases, depending on the capabilities of the particular DBMS. For example, DB2 and INGRES can both manage more than one application database per DBMS installation, whereas ORACLE will only manage one. The internal data are usually referred to as the *data dictionary* and used by the DBMS to properly manage the application data in the system. In [19], ORACLE defines the data dictionary as a set of views and tables that contain information about the definitions of objects in the database. DB2 refers to its data dictionary as the DB2 catalog. The DB2 catalog is comprised of tables of data in a system-defined database that contain information about everything defined to the DB2 system (see Chapter 1 of [13]). Finally INGRES refers to its data dictionary as the INGRES System Catalogs. These catalogs are tables and views that contain information about every database and database object in the system.

The concept of the data dictionary is not defined by the ANSI SQL standard. Nevertheless, the data dictionary is necessary for the practical operation of a commercial DBMS. As a result, each DBMS has its own particular nonstandard data dictionary implementation.

### Database Declaration and Evolution

The following two sections discuss typical database definition semantics for commercial DBMS implementations.

### Database Declaration

The semantics of initial database declaration in commercial DBMS implementations differ from the semantics specified in ANSI standard SQL. In the standard, the tables, views, and privileges of a schema are defined at the time that the schema is defined. Because database evolution is not addressed in the standard, schema elements cannot be added to a schema once that schema has been defined.

In commercial implementations, database objects cannot be defined at the time that the database itself is defined. For example, the ORACLE database exists when the ORACLE DBMS has been installed. In a DB2 installation, a default database already exists after installation of the DBMS, and subsequent databases are created by the nonstandard *CREATE DATABASE* SQL statement (see Chapter 4 of [12]). The execution of this statement causes the creation of an empty database in the DB2 DBMS. In INGRES, the DBA tool *Createdb* (see Chapter 4 of [15]) similarly creates an empty database in an INGRES DBMS. Because of the above semantics, the initial declaration of tables, views, and other database objects in a commercial DBMS implementation is a special case of database evolution.

#### 3.2.2.2. Database Evolution

As described in the introduction, database evolution is the incremental modification of a database definition, after its initial declaration. It is the creation, alteration, and deletion of database objects such as tables, views, indexes, and procedures. In commercial DBMS, because a database is initially created without database objects, initial database declaration is a special case of evolution. It is simply the creation of the initial set of database objects in a database.

Databases in commercial DBMS are used to partition application data such that they are logically grouped by application, and access can be controlled. Database evolution in a commercial DBMS, therefore, occurs for a specific database. Since commercial DBMS have the ability to manage more than one database at a time, however, DBMS vendors each provide a mechanism for specifying the database for which the evolution is intended. For example, the DB2 DBMS provides SQL syntax that allows the DBAN to qualify a database object's name with the name of the database when creating, altering, or dropping that object (see Chapter 4 of [12]). INGRES, however, provides two SQL statements not found in the ANSI standard that allow an application to be attached to a specific database within the system. These SQL statements are *CONNECT* and *DISCONNECT*. Once the DBA or application is connected to a particular database, any execution of SQL including DDL will affect only that database (see Chapter 8 of [14]). ORACLE also provides a *CONNECT* and *DISCONNECT*, even though there is only one database in the DBMS (see Chapter 2 of [19]).

#### 3.2.2.3. Commercial DDL Implementations

Commercial DBMS implementations generally extend standard SQL with features for database evolution. These extensions fall into three general categories.

1. Creation and maintenance of performance oriented structures. These are variously called *clusters, indexes, partitions, spaces*, etc. These structures have no effect on application program semantics and are therefore not covered by the ANSI SQL standard.
2. Support for data types not provided in the ANSI SQL standard. A calendar or *date* data type is the most popular extension of the standard.
3. Modification to and deletion of existing structures. The nonstandard statement *DROP TABLE* is probably found in every SQL implementation. The nonstan-

dard statement *ALTER TABLE*, which adds or deletes columns in an existing table, appears in both ORACLE and DB2 (see Chapter 2 of [19] and Chapter 4 of [12], respectively.).

It may well be wise for a SAMeDL processor to ignore the performance-oriented DDL statements, since those statements have no effect on the meaning of any application. On the other hand, commercial DBMS allow those DDL statements in applications, primarily for the use of system software tools. If such tools are to be written in Ada, support for those statements should appear in the SAMeDL processor.

The SAMeDL supports nonstandard data types through user-defined base domains (see Section 4.1 of [8] and Section 5.2 of [9]). There is no need to extend the language, provided that the type of the data *at the concrete interface* (see [11]) is one of those in SQL_Standard. In ORACLE, INGRES, and DB2, calendar data appear as character strings at the concrete interface.

SAMeDL support for the DDL extensions *DROP TABLE* and *ALTER TABLE* is discussed in Section 3.3.2 below.

#### 3.2.2.4. Executable DDL

Most commercial DBMS allow some subset of their DDL to be embedded in one or more programming languages to provide an executable mechanism for database definition. This facilitates the construction of database definition system software tools by parties other than the DBMS vendor. More importantly, however, it provides application programmers with the means to create and delete temporary tables, as described in the introduction.

DB2, INGRES, and ORACLE all allow their complete sets of DDL statements to be embedded in application programs (see Chapter 4 of [12], Chapter 8 of [14], and Chapter 1 of [17], respectively).

#### 3.2.2.5. The Data Dictionary

Whenever a database object is defined, altered, or deleted in a commercial DBMS, system-specific information about that object is either added, updated, or deleted from the data dictionary tables, respectively. The data dictionary provides the DBMS with a mechanism for recording the attributes and status of database objects that exist in the databases under its control.

The data dictionary is immediately updated when a database object is created, modified, or deleted by batch or interactive system software tools that process non-executable DDL. On the other hand, executable DDL has its effect on the data dictionary at the time the application program is executed. However, because executable DDL is precompiled, some DBMS implementations perform updates to the data dictionary prior to the actual creation of the database object. For example, DB2 has a *BIND* step that must be performed before an application program containing executable DDL can be executed (see Chapter 4 of [12]). This step verifies the existence of tables and columns used in the embedded SQL statements and checks that the application has authorization to execute the statement. Most

DBMS implementations, however, do not have a BIND step and make updates to the data dictionary at runtime when the DDL statements are actually executed.

## 3.2.3. SAMeDL Database Definition

The SAMeDL is designed to support the creation of Ada application programs that access data in commercial SQL DBMS. The following section is a discussion of the terminology and semantics of database definition in the SAMeDL.

### 3.2.3.1. Terminology

The primary concept of database definition in the SAMeDL is the *schema module*. The schema module is a logical grouping of tables, views, and privileges. These schema module elements are modeled after the schema elements of ANSI standard SQL.

SAMeDL schema modules, however, are not modeled after ANSI standard SQL schemas, and they do not have similar semantics. In fact, the SAMeDL language definition (see [8]) remains silent on the SQL semantics of the SAMeDL schema module. Schema modules simply provide a mechanism with which to declare schema elements such as tables, views, and privileges, and a way to logically group these objects for use by SAMeDL abstract modules (see Chapter 5 of [8]) and other schema modules (see Section 4.2 of [8]).

The environment in the SAMeDL is implementation defined, as it is in ANSI standard SQL. However, the environment is assumed to contain information about schema module and definitional module elements that have been previously processed by the SAMeDL compiler. Such information makes it possible for the SAMeDL compiler to perform compile time type checking of procedure and cursor declarations in a SAMeDL abstract module.

### 3.2.3.2. Database Declaration and Evolution

Because the SAMeDL does not discuss schemas or databases, or any other means for partitioning the data in the SQL environment, the language definition (see [8]) makes no correlation between these objects and SAMeDL schema modules. Therefore, SAMeDL schema elements are defined incrementally and logically grouped to support application development. All database definitions in the SAMeDL, then, are essentially evolutionary. Even initial database declaration is considered to be just the first increment, and a special case, of a constantly evolving SAMeDL database definition.

### 3.2.3.3. Executable DDL

As specified in the language definition (see [8]), the SAMeDL does not define syntax for executable DDL. The SAMeDL compiler implementor may, however, implement language extensions to provide the SAMeDL programmer with this capability (see Section 3.7 of [8] and Chapter 3.3 below).

#### 3.2.3.4. The Data Dictionary

As described above in Subsection 3.2.3.1, the SAMeDL environment contains information about schema module and definitional module elements that have been previously processed by the SAMeDL compiler. SAMeDL database definition provides the DBA with the means to update the environment with this information. The SAMeDL compiler later uses this information when compiling subsequent definitional, abstract, and schema modules to verify the existence of schema elements and to perform the appropriate domain checking in the declarations. The information described above can be stored in a SAMeDL data dictionary (or catalog tables) similar to those described above for commercial DBMS implementations.

## 3.3. Extending the SAMeDL

As described above in Section 3.2.3 database definition in the SAMeDL is essentially evolutionary. The SAMeDL, however, specifies only the *creation* of schema elements in the language definition (see [8]). It does not address the modification and deletion of schema elements because the DDL of the SAMeDL is modeled after the DDL of ANSI standard SQL. Commercial DBMS, however, do implement DDL that performs modification and deletion of database objects. These DBMS also provide DDL that allows the creation of database objects that have no ANSI standard SQL equivalent.

Furthermore, the SAMeDL does not address the issue of executable DDL. The SAMeDL language definition remains silent on the issue because executable DDL is not addressed by the ANSI standard. Commercial DBMS do, however, provide executable DDL (see Subsection 3.2.2.4 above) because it can be a valuable tool with which programmers can perform query optimizations (see the introduction).

To support both executable DDL and DDL that performs database evolution, the SAMeDL compiler implementor must provide SAMeDL language extensions. The following sections describe examples for extending the language.

### 3.3.1. Extended Schema Elements

All SAMeDL statements, including language extensions, that modify the SAMeDL data dictionary are considered to be DDL and must be implemented as *schema elements* (see Section 4.2 of [8]). Any schema element that does not have identical syntax to the *table definition*, *view definition*, or *SQL privilege definition* of the SAMeDL must be implemented as an *extended schema element* (see Section 3.7 of [8]). Therefore, any SAMeDL language extensions that are implemented to support the modification and deletion of SAMeDL schema elements, the creation of schema elements not specified in the language definition (see Section 4.2 of [8]), and executable DDL (see Section 3.3.3 below) must be implemented as an extended schema element in the schema module.

As described in [9], extensions must maintain the syntactic and semantic style of the SAMeDL so as to minimize portability problems. The syntax and semantics of SAMeDL extended schema elements must adhere to these guidelines. For example, the syntax for the extended schema elements *must* start with the SAMeDL language keyword **extended**. This keyword labels the DDL statement as a possibly nonportable language extension. Furthermore, extensions must remain strongly typed. Column definitions in extended schema elements, for example, must contain domain references to maintain the strong type-checking semantics of the SAMeDL.[18] Furthermore, as a general guideline, the language definition for the SAMeDL (see Section 3.7 of [8]) states that "any portion of an extension whose semantics may be expressed in standard SAMeDL, shall be expressed in standard SAMeDL syntax."

### 3.3.2. Database Evolution

As described above in Paragraph 3.2.2.2, all commercial DBMS implementations provide DDL that performs the necessary task of database evolution. The ANSI draft proposed follow-on standard for SQL, called SQL2 (see [5]), also recognizes the need for database evolution and provides SQL statements that modify and delete schema elements.

The SAMeDL can be extended to support database evolution. The syntax for these SAMeDL language extensions can be based on either the syntax of the equivalent SQL statement in the target commercial DBMS or the syntax of the equivalent SQL2 statement. There are advantages to both; however, it is recommended that the syntax follow that of the SQL2 statement.

If the syntax is based on the SQL statement of the commercial DBMS, then the language extension can be identified as belonging to a particular DBMS implementation. The syntax of the statement itself identifies it as nonstandard.

There is a major advantage to providing syntax for the language extension that is identical to the SQL2 statement, however. If SQL2 should become a standard, the SAMeDL would possibly be redefined to be based on SQL2 rather than SQL. If this should happen, the language extension would be incorporated in its entirety and without modification (except the word **extended** would be dropped from the syntax) into the language definition. There would be no need for significant changes to the SAMeDL processor to provide this formerly *extended* SAMeDL statement.

This section illustrates examples of language extensions that can be added to the SAMeDL to support database evolution. The examples below show both SQL2 syntax as well as commercial SQL syntax versions of the language extension. These examples are intended to provide the SAMeDL compiler implementor with samples of possible implementations of language extensions. The SAMeDL compiler implementor is free to extend the language to support any DDL statement found in a commercial implementation, or in SQL2.

---

[18] See the rules for extending value expressions, search conditions, input parameter lists, table elements, select parameter lists, and into clauses, as specified in Sections 5.10, 5.11, 5.6, 4.2, 5.7, and 5.9 of [8], respectively.

## Examples

The following is an example of an *ALTER TABLE* SAMeDL language extension. The example alters a table to add a column, using syntax that resembles the SQL2 *ALTER TABLE* statement.

```
        extended ALTER TABLE Parts ADD
                     Part_Description    CHAR(25) : Part_Description_Domain;

        extended ALTER TABLE Parts ADD
                     Part_Catalog_Number CHAR(10) : Catalog_Number_Domain;
```

In the above example, the syntax is SAMeDL syntax, not SQL2 syntax, although it resembles SQL2. The column definition is identical to the syntax found in the current SAMeDL language definition.

Below is a version of SAMeDL syntax that resembles the syntax found in the ORACLE SQL implementation.

```
        extended ALTER TABLE Parts ADD (
            Part_Description    CHAR(25) : Part_Description_Domain,
            Part_Catalog_Number CHAR(10) : Catalog_Number_Domain);
```

Using the ORACLE syntax, one *ALTER TABLE* statement is sufficient to define both columns, whereas with SQL2 syntax two statements are required. Notice also that the column definition is identical to the syntax of the current SAMeDL column definition.

Below is an example of SAMeDL syntax for the *DROP TABLE* statement based on SQL2 syntax.

```
        extended DROP TABLE Parts;
```

SQL2 also provides a *CASCADE* option on the *DROP* statement that is not reflected in the above example.

**NOTE:** A SAMeDL language extension for *DROP TABLE* may not be necessary. The effect of dropping a table can be achieved by the recompilation of the defining schema module minus the original *CREATE TABLE* statement that originally defined the table. Alternatively the SAMeDL compiler implementor may also provide compiler options that "maintain" the environment by allowing the user to drop table definitions and other unused schema elements in the SAMeDL environment.

### 3.3.3. Executable DDL

All SAMeDL DDL *must* appear within schema modules, as there is no mechanism to reference tables or views except when they are declared within such modules. Therefore, a SAMeDL processor supporting executable DDL may create Ada and SQL texts for (some) schema modules in much the same way it does for abstract modules. In other words,

- Corresponding to each schema module containing executable DDL, there may be an Ada package whose package name is the schema module name.
- Within that Ada package there is an Ada subprogram declaration for each ex-

ecutable DDL statement in the schema module. Execution of that subprogram by the Ada application affects the data description specified by the DDL statement.

It is possible to support executable DDL *without* any language extensions, or, at least, without any extensions above and beyond those needed to support schema evolution from application programs, e.g., *ALTER TABLE*. A SAMeDL processor is free to create the Ada and SQL texts mentioned in (or implied by) the list above for any and all schema modules.[19] However, such support does not offer the SAMeDL programmer the opportunity to name the generated Ada subprogram or to attach a status map to the procedure. Therefore it may be better to create SAMeDL extensions for executable DDL. One such proposal is the following syntax.

```
<extended_schema_element> ::=

    extended procedure Ada_Identifier [<input_parameter_list>] is
                <schema_element>
                [<status_clause>];
```

An example of such an extension is the extended executable create temporary table statement in the following schema module.

```
schema module Temporary_Tables is

    extended procedure Create_Temporary_Parts_Table is
        table Temporary_Parts_Table is
            Part_Number : Part_Number_Domain;
            Part_Name   : Part_Name_Domain;
        end Temporary_Parts_Table
        status Standard_Map named Successful_Create;

end Temporary_Tables;
```

It is probably good programming practice to restrict schema modules containing executable DDL to contain only executable DDL. Of course, it is also good practice to ensure that the collection of statements in a module is logically coherent. The **create** and **drop** for a temporary table should appear together and probably alone in a schema module.

**NOTE**: Grouping temporary table definitions in schema modules in this way, and logically by application, the SAMeDL programmer need no longer worry about table name collisions for temporary tables since the tables are referenced by schema module name and table name.

[19]It might be wise for the compiler vendor to supply a runtime switch to control this behavior.

## 3.4. Data Dictionary Consistency

The introduction of a data dictionary in the SAMeDL environment creates a new problem for the SAMeDL compiler implementor. The SAMeDL data dictionary and the one implemented for the DBMS must be kept consistent. The two issues that must be addressed are:

1. The definition of the term *data dictionary consistency*.
2. How and to what extent consistency can be maintained.

The two data dictionaries are said to be consistent if each has identical descriptions for individual database objects such as tables and views. For example, two descriptions for the same table must have the same table name and the same number of columns, with identical names, in the same order. The two descriptions need only be identical for the common information about a database object; each data dictionary will contain information that is specific to its respective environment. For example, a description of a table in a SAMeDL data dictionary will contain a domain reference (see [11]) for each column, whereas a description in the DBMS data dictionary will contain the data type specification instead. Consistency also means that there must be a description in each data dictionary for every existing database object.

Consistency between data dictionaries must be maintained when database definition is accomplished either by application programs or by system software utilities. Each mechanism for database definition has different effects on the data dictionaries.

When using vendor-supplied system software utilities to define, alter, or delete database objects such as tables and views, the DBMS data dictionary is immediately updated to reflect the change. Similarly, the SAMeDL data dictionary is immediately updated when the SAMeDL compiler processes non-executable DDL[20] in SAMeDL schema modules. The fact that two sets of DDL are required (SAMeDL and commercial SQL) to update the data dictionaries for any one definition, alteration, or deletion of a database object means that the two data dictionaries will be inconsistent for a time because the two sets of DDL cannot be processed simultaneously. One approach to solving this problem is to implement a SAMeDL compiler that processes the DDL in the SAMeDL schema module first, and then creates an operating system process or job to invoke the database definition system software utilities of the DBMS to update the DBMS data dictionary. A second approach is to implement a SAMeDL compiler that updates *both* data dictionaries simultaneously, performing the database definition in the DBMS as well. This approach, however, may require the SAMeDL compiler implementor to have detailed and possibly proprietary knowledge of the DBMS. Therefore it is much more feasible if the SAMeDL compiler implementor and the DBMS vendor are the same in this case. The goal of both of these approaches is to define, alter, or delete database objects in both data dictionaries simultaneously.

---

[20]Non-executable means *not* embedded in, and therefore executed from within, an application program. See Section 3.3.3 above for details on supporting executable DDL in the SAMeDL.

When defining database objects from within application programs (i.e., through executable DDL), maintaining data dictionary consistency is a much more complex issue because the two data dictionaries must be inconsistent for some period of time. There are different semantics, however, for the various types of DDL. First, consider the creation of database objects. Below is an example of an extended procedure for creating a temporary table.

```
schema module Temporary_Tables is

    extended procedure Create_Temporary_Parts_Table is
        table Temporary_Parts_Table is
            Part_Number : Part_Number_Domain;
            Part_Name   : Part_Name_Domain;
        end Temporary_Parts_Table
        status Standard_Map named Successful_Create;

    extended procedure Drop_Temporary_Parts_Table is
        DROP TABLE Temporary_Parts_Table;

end Temporary_Tables;
```

When executable DDL statements (see Section 3.3.3 above) that define tables and views in the SAMeDL are compiled in a schema module, the SAMeDL data dictionary must be updated immediately, even though these tables and views will not be defined in the DBMS data dictionary until the corresponding executable DDL statements are executed from within an application program. The SAMeDL data dictionary must be updated immediately because the descriptions of these tables and views must be known to the SAMeDL data dictionary when subsequent schema and abstract modules that contain references to these objects are compiled. To further illustrate this, consider the abstract module below.

```
abstract module Use_Temporary_Tables is
    authorization Temporary_Tables

    extended procedure Select_From_Temporary_Parts_Table is
        select Part_Number, Part_Name
        from Temporary_Parts_Table
        where Part_Number = 'A104'
        status Standard_Map;

end Use_Temporary_Tables;
```

To perform the necessary type checking on procedure *Select_From_Temporary_Parts_Table* the SAMeDL compiler must know of the existence of table *Temporary_Parts_Table* with columns *Part_Number* and *Part_Name*, and that the Part_Number column contains character data. This requires that the *Temporary_Tables* schema module be compiled by the SAMeDL compiler first, and that the SAMeDL data dictionary retain a description of the table, even though the DBMS data dictionary will not be updated until the *Create_Temporary_Parts_Table* procedure is executed later. It can be seen that the SAMeDL and DBMS data dictionaries must remain inconsistent for a time.

Now consider the semantics for dropping database objects. The compilation of the extended procedure *Drop_Temporary_Parts_Table* in schema module *Temporary_Tables* above must not delete the description of the table *Temporary_Parts_Table* from the SAMeDL data dictionary because the description is created by the compilation of the previous procedure, and references to it from procedures in subsequently compiled schema and abstract modules must still be resolved. Therefore, besides generating executable code, the compilation of this statement by the SAMeDL compiler will result in a check that a definition for *Temporary_Parts_Table* already exists in the SAMeDL data dictionary.

The issue that must be resolved in this case is *when* to delete the description of *Temporary_Parts_Table* from the SAMeDL data dictionary, since execution of the corresponding procedure *Drop_Temporary_Parts_Table* by the application will only remove the description from the DBMS data dictionary, leaving the two data dictionaries inconsistent. In one approach, the SAMeDL compiler implementor can provide a compiler option, or an additional SAMeDL tool for the environment, to remove these definitions from the SAMeDL data dictionary. In another approach, the SAMeDL compiler implementor can provide the SAMeDL programmer with the means to recompile a schema module. In this case, recompilation of the schema module, minus the procedure *Create_Temporary_Parts_Table*, would have the effect of removing the description of *Temporary_Parts_Table* in the SAMeDL data dictionary.

# 4. Support for Multiple Concurrent Transactions in the SAMeDL

**by Gregory Zelesnik**

## 4.1. Introduction

Ada tasks provide application programmers with the capability of executing multiple independent threads of control simultaneously within one application program. The tasking capability of Ada allows application programs to perform separate units of work in parallel. For example, in process control applications such as avionics software, tasks are used to monitor many different sensors simultaneously, regularly reporting their status in real time (see Section 6.3 of [6]).

Many application development organizations wish to take advantage of Ada tasking in applications that access data from SQL databases to perform multiple transactions[21] in parallel. The problem is that neither ANSI standard SQL nor the follow-on standard, SQL2 (see [5]), allows for the execution of multiple concurrent transactions from within one application program. This is because there is no way in the standard to associate the execution of an individual database operation with a particular transaction among a concurrent set of transactions.

Some commercial database management systems provide support for execution of multiple concurrent transactions from within a single application program;[22] SYBASE, INGRES, Interbase, and Unify 2000 are examples of such DBMS. Other commercial DBMS are designed to support only sequential transaction processing from within a single application program.[23] While it may be possible to create application programs that execute multiple concurrent transactions in the latter case,[24] it is recommended that any applications that process multiple concurrent transactions be built on a DBMS platform that directly supports them (see Section 4.2.1).

This chapter describes alternative strategies by which a SAMeDL compiler might support multiple concurrent transaction applications.

---

[21]A transaction is a sequence of database operations that is atomic with respect to recovery and concurrency (see Section 4.16 of [4]). It is the basic unit of work in a DBMS application program.

[22]Throughout the rest of this chapter, the term *multiple concurrent transactions* will be used to refer to multiple concurrent transactions executing from within a *single* application program.

[23]The list of commercial DBMS presented here is not meant to be an endorsement of the systems mentioned. Nor is it intended to represent a complete list of those DBMS that support multiple concurrent transactions within a single application program.

[24]E.g., by associating a separate operating system process with each Ada task so that the commercial DBMS perceives the single application as multiple processes.

## 4.2. Multiple Concurrent Transactions

The ability to construct application programs that execute multiple concurrent transactions depends upon:

- The application support code of the target DBMS.
- The ability of the DBMS to identify transactions.

Each of these is discussed in greater detail below.

### 4.2.1. The Target DBMS

The target DBMS must be able to support multiple concurrent transactions executing from within a single application program. Although it may be possible to construct application programs that process multiple concurrent transactions for a target DBMS that does not support this functionality, it is recommended that this not be attempted. Any such applications would most certainly rely heavily upon:

- operating system primitives
- an unsupported DBMS interface

or some combination of the above. This would make the application non-portable, and maintenance would become unmanageable as new versions of the target DBMS and operating system are released.

To support multiple concurrent transactions, a DBMS must be able to interact with the multiple threads of control in the application simultaneously. This support must be built into the DBMS application support code that is incorporated into each application. Every DBMS application program will contain some code supplied by the DBMS as part of its executable image. This code is typically supplied with the DBMS in object code libraries, and it is incorporated into the application during the link step of application development. The function of the DBMS code is to accept the call from the application program, and transfer the call to the DBMS, which is usually executing as a separate process on the same machine or on another machine in a network.

Traditionally DBMS applications have executed as a single thread of control, having been written in programming languages other than Ada. Only one copy of the DBMS code has been necessary, because calls to the DBMS have been serial in this case. Ada tasking, however, creates a new problem for the DBMS by introducing multiple threads of control in a single application. In this case it is possible for two Ada tasks to make DBMS calls simultaneously, requiring access to the DBMS code at the same time. To support multiple threads of control in a single Ada application, either:

- The DBMS code must be reentrant.
- Or each thread of control must have its own private copy of the DBMS code.

Either of the above conditions must hold, or the multiple threads of control will collide within the DBMS code, and the results of both DBMS calls will be undefined.

### 4.2.2. Identification of Transactions

If the reentrancy requirement of the above section is met by the target DBMS, the next requirement is for the DBMS to provide a means for identifying individual transactions. This is necessary so that the execution of a particular database operation can be associated with one of the active transactions.

To distinguish one transaction from another in an application, the transactions must be labeled. The approach used for labeling transactions depends upon the particular model for processing multiple concurrent transactions implemented by the DBMS.

The most common way that DBMS implement multiple concurrent transactions is to allow an application program to simultaneously maintain multiple user connections to one or more databases in the DBMS. Within each user connection, then, sequential transaction processing is strictly enforced, meaning that only one transaction can be active in a user connection at any one time. Using this model for processing multiple concurrent transactions, the DBMS labels the transaction by labeling the user connection. In effect, the user connection label is the transaction identifier.

Another model that DBMS may use to implement multiple concurrent transactions is to allow an application program to simultaneously process multiple transactions from within a single user connection.[25] Using this model, the DBMS must label each individual transaction. This label must, then, be supplied as a parameter to each DBMS call so that the database operation can be associated with a particular transaction.

In both of the above models, the transaction identifier is usually a pointer to a *communication area*. Every transaction has an associated communication area, a storage area maintained by the DBMS. It is used by the application to pass information to the DBMS about a database operation. It is also used by the DBMS to pass results from, and execution status of, the database operation.

## 4.3. SAMeDL Language Extensions

As described above, ANSI standard SQL does not support the execution of multiple concurrent transactions from within a single application program. This is because the standard does not provide the application programmer with a means to associate the execution of a database operation with a particular transaction. Commercial DBMS, however, do support multiple concurrent transactions and provide ways to identify transactions.

As discussed in Section 4.2.2, a DBMS will allow the application program to either establish multiple user connections in which sequential transaction processing is performed, or initiate multiple concurrent transactions from within an individual user connection. To support multi-

[25]Although this model is theoretically possible, the author knows of no such DBMS that implements it.

ple user connections, a DBMS must provide an SQL statement whose purpose is to establish a new distinct user connection. An example of such a statement is the *CONNECT* statement. Many commercial DBMS include such a statement in their implementations of SQL. To support multiple concurrent transactions from within a single user connection, a DBMS must provide an SQL statement whose purpose is to initiate a new distinct transaction. An example of such a statement might be a *BEGIN TRANSACTION* statement.

Neither the *CONNECT* nor the *BEGIN TRANSACTION* statement is part of ANSI standard SQL, and consequently not part of the SAMeDL either. To provide support for them in the SAMeDL, therefore, they must be implemented as language extensions.

There are two issues that must be addressed when considering the implementation of SAMeDL language extensions for these statements:

1. Whether or not any extended statements are necessary.
2. What transaction processing model should be used.

**NOTE**: It must be noted here that either or both of the *CONNECT* and *BEGIN TRANSACTION* statements described above are most likely required to support multiple concurrent transactions, and may be implemented as SAMeDL extensions. It may not be necessary to extend any other SAMeDL statements, unless support for a transaction identifier in the statement causes the SAMeDL syntax to deviate from its specification in [8] (see Section 4.4.2 below).

### 4.3.1. Extending the SAMeDL

Before implementing a language extension in the SAMeDL for a *CONNECT* or a *BEGIN TRANSACTION* statement, the SAMeDL compiler implementor must decide whether or not an extension is actually necessary. If either or both of these statements can be completely parameterized, then a SAMeDL programmer will never have to provide more than one *CONNECT* or *BEGIN TRANSACTION* procedure in a single application. For example, in the case of the *CONNECT* statement, if the database name can be parameterized, then the application program can use the same procedure for each user connection.

If only one procedure is ever necessary in any application, then no SAMeDL language extension is required. The *CONNECT* or *BEGIN TRANSACTION* procedure can be supplied by the SAMeDL environment, and *with*ed into the Ada application program.

If, however, these statements require values that cannot be parameterized, then they must be supported in the SAMeDL with a language extension because a different *CONNECT* statement is required for every database connection.

For an example of a *CONNECT* procedure, see Section 4.4.3 below.

### 4.3.2. Transaction Processing Model

If SAMeDL language extensions are required, it is necessary for the SAMeDL compiler implementor to choose the appropriate transaction processing model (see Section 4.2.2 above). The issue here is whether or not to directly support the model of the underlying target DBMS, or to support a more general one. The type of model chosen will have an impact on which statements are implemented as extensions, and what their syntax will be.

If the SAMeDL compiler implementor and the DBMS vendor are the same entity, then the model for the SAMeDL will most likely be the same as the one for the DBMS. In this case the extended SAMeDL statements and their syntax would look similar to the underlying SQL statements of the DBMS platform. If the SAMeDL compiler implementor is a third-party vendor, and the compiler is being developed for multiple platforms, the model of transaction processing will be more general. The extended SAMeDL statements and their syntax may not resemble those of any one particular DBMS.

## 4.4. SAMeDL Compiler Enhancements

This section describes implementation issues with respect to support of multiple concurrent transactions in the SAMeDL.

### 4.4.1. The Transaction Identifier Data Type

Support for multiple concurrent transactions implies support for transaction identifiers (see Section 4.2.2 above). Transaction identifiers are necessary for identifying the particular transaction on behalf of which a database operation is executed. The Ada application program is responsible for declaring and maintaining transaction identifiers, and using them to properly construct DBMS transactions.

Transaction identifiers exist as objects in the Ada application program, so they must have a data type. The Ada type of these objects is a private type supplied with the SAMeDL compiler, since the underlying concrete representation of the transaction identifier is DBMS specific and the application programmer has no need to update transaction identifier objects. These objects are obtained by the application from the DBMS and passed to and from the DBMS for transaction identification only.

Transaction identifiers may or may not appear explicitly in the text of a SAMeDL procedure (see Section 4.4.2.3 below). If they do, then they must have an associated SAMeDL domain, the definition of which is supplied with the compiler in a definitional module with a predetermined name, e.g., <*vendor_name*>_Domains. The SAMeDL programmer will **with** the module to access the domain, and the Ada programmer will **with** the derived Ada package to access the derived Ada types. If transaction identifiers do not appear in SAMeDL text, the Ada package and type declaration are still needed but can be supplied directly.

### 4.4.2. Transaction Identifier Parameters

This section discusses transaction identifier parameters in the parameter profile of the Ada procedures in the *abstract* interface of an Ada application. The discussion assumes a model of transaction processing that requires a transaction identifier parameter in every database procedure call at the *concrete* interface.[26] We give three alternatives by which the decision to include or omit a transaction identifier parameter at the abstract interface can be made.

1. The parameter appears in the profile of every procedure in every abstract interface.
2. The parameter appears in the profile of every procedure in every abstract interface of a given SAMeDL compilation (a compiler option).
3. The parameter will only appear in the profile of individual procedures that explicitly request it. (We give two alternatives for this case.)

These approaches are discussed below.

#### 4.4.2.1. Every Procedure

Using this approach there is an Ada formal parameter declaration for a transaction identifier object in *every* Ada procedure declaration. The formal parameter has a predetermined name, such as *Transaction_Identifier*, and is declared to be of the Ada type described in Section 4.4.1 above.

With this approach a transaction identifier parameter is generated for every procedure at the interface, whether or not the application programmer intends to process multiple concurrent transactions. To make life easier for those application programmers processing one transaction at a time, the above described *Transaction_Identifier* formal parameter is supplied with a *default_expression* (see Section 6.1 of [3]). This default expression names a default transaction identifier object, supplied by the SAMeDL compiler vendor in the same Ada package in which the transaction identifier Ada data type is declared (see Section 4.4.1 above). The default transaction identifier has a predefined name, such as *Default_Transaction*.

To write Ada application code that processes transactions one at a time, the Ada programmer need only supply the default transaction identifier object to the *CONNECT* statement to accept the transaction identifier from the DBMS for the first time.[27] Thereafter, the Ada programmer need not specify a transaction identifier to any DBMS call. The Ada procedures at the abstract interface will just use the default transaction identifier in each case.

[26] INGRES Version 6.3 for Ultrix implements a model for processing multiple concurrent transactions that does not require a transaction identifier for each database call. Multiple user connections can be made within a single INGRES application, but only one is active at any one time. Database operations are executed against the active user connection. Therefore, no transaction identifier, or user session identifier in this case, is required when a database call is made. The session identifier is only required to make another user connection active.

[27] Another alternative is for the SAMeDL compiler vendor to supply the application programmer with a predefined *connect* procedure that does this, without having an associated SAMeDL *CONNECT* statement in the language.

To write Ada application code that processes multiple concurrent transactions, the Ada programmer declares multiple transaction identifier objects in the application program, then supplies the appropriate transaction identifier to each call to the DBMS. In this case, the default transaction identifier object is never used.

The advantage of using this approach is that, for those application programmers who wish to process only one transaction at a time, the use of the *Default_Transaction* transaction identifier makes writing the application program easier. For those DBMS that require transaction identifiers with every DBMS call, a transaction identifier is still required in the case where the programmer is processing one transaction at a time. With this approach, the programmer does not have to specify transaction identifiers with the DBMS calls, and the *Default_Transaction* identifier is supplied by the procedure at the abstract interface.

The disadvantage of using this approach is that if an application programmer is processing multiple transactions concurrently and forgets to supply a transaction identifier for a particular DBMS call, then the default transaction identifier will be used. The application program will successfully compile and may execute without error, yet the mapping of the database calls to the transactions will have been erroneous. This will create erroneous data in the DBMS, and the problem may go undetected. Even if the problem is detected, debugging of such an application is very difficult.

## Example

Consider the following SAMeDL abstract module.

```
with Parts_Domains;  use Parts_Domains;
abstract module Parts_Module is
    authorization Parts_Suppliers_Database

    procedure Delete_Parts (
        Input_Pname named Part_Name : Pname) is

        delete from P
            where PNAME = Input_Pname;

end Parts_Module;
```

Using the above mentioned approach, the Ada code at the abstract interface looks something like the following.

```
with <vendor_name>_System_Package;
with Parts_Domains; use Parts_Domains;
package Parts_Module is

    procedure Delete_Parts (
        Part_Name              : Pname_Type;
        Transaction_Identifier :
            <vendor_name>_System_Package.Transaction_Identifier_Type :=
            <vendor_name>_System_Package.Default_Transaction_Identifier);

end Parts_Module;
```

Notice that the second parameter to the Ada procedure is the *Transaction_Identifier* parameter. The default expression names the global *Default_Transaction_Identifier* that exists in the Ada package *<vendor_name>_System_Package*, provided with the SAMeDL compilation system.

#### 4.4.2.2. SAMeDL Compiler Option

Using the SAMeDL compiler option approach, the SAMeDL programmer controls whether the transaction identifier parameter appears in Ada procedures at the abstract interface on a compilation by compilation basis.

When the compiler option is turned on for a compilation, every generated Ada procedure contains a formal parameter declaration for a transaction identifier object. The formal parameter has a predetermined name, such as *Transaction_Identifier*, and is declared to be of the Ada type described in Section 4.4.1 above. In this case, there is no default expression for this parameter declaration. This requires the Ada application programmer to supply a transaction identifier for each call.

When the compiler option is turned off for a compilation, there is no formal parameter declared in any Ada procedure for a transaction identifier object. The corresponding Ada procedure *body* in the abstract module (see Section 4.4.3 below) uses a default transaction identifier object, similar to the one described in the second paragraph of Section 4.4.2.1 above.

The advantage of having a SAMeDL compiler option is that code generation for multiple concurrent transactions can be controlled. The abstract interface for applications that process transactions one at a time is much simpler, and can prevent less experienced Ada application programmers from getting into trouble.

With a SAMeDL compiler option such as this, however, there is the potential for constructing an application program that processes multiple concurrent transactions with abstract modules that have been compiled both ways (compilations compiled with *and* without the compiler option turned on). This is dangerous. If the application programmer is processing multiple concurrent transactions, then *all* the abstract modules should be compiled with the SAMeDL compiler option turned on.

### Example

Again, consider the following SAMeDL abstract module.

```
with Parts_Domains;  use Parts_Domains;
abstract module Parts_Module is
    authorization Parts_Suppliers_Database

    procedure Delete_Parts (
        Input_Pname named Part_Name : Pname) is

        delete from P
            where PNAME = Input_Pname;
```

```
end Parts_Module;
```

Using the above-mentioned approach with the compiler option *turned on*, the Ada code at the abstract interface looks something like the following.

```
with <vendor_name>_System_Package;
with Parts_Domains; use Parts_Domains;
package Parts_Module is

    procedure Delete_Parts (
        Part_Name              : Pname_Type;
        Transaction_Identifier :
            <vendor_name>_System_Package.Transaction_Identifier_Type);

end Parts_Module;
```

Notice that it is almost identical to the Ada code fragment in Section 4.4.2.1 above, except that the *Transaction_Identifier* formal parameter has no default expression in this case.

If the compiler option is *turned off*, the following code fragment is produced.

```
with Parts_Domains; use Parts_Domains;
package Parts_Module is

    procedure Delete_Parts (Part_Name : Pname_Type);

end Parts_Module;
```

Notice that in this case no *Transaction_Identifier* formal parameter is generated. The *Default_Transaction_Identifier* is used, but is global to the procedure body in the package body. It is not passed in as a parameter to the procedure. Consequently, the package *<vendor_name>_System_Package* is *with*ed into the *Parts_Module* package body.

### 4.4.2.3. Individual Procedures

We present two ways in which a SAMeDL procedure might specifically request a transaction identifier parameter. The first method is via an extension, as in the following example:

```
with Parts_Domains;  use Parts_Domains;
extended abstract module Parts_Module is
    authorization Parts_Suppliers_Database

    extended multitrans procedure Delete_Parts (
        Input_Pname named Part_Name : Pname) is

        delete from P
            where PNAME = Input_Pname;

end Parts_Module;
```

In the above example, the keyword **multitrans** is used to designate that an individual procedure is to be used by multiple transactions. Upon detecting the keyword, the compiler generates an Ada procedure at the abstract interface that includes a transaction identifier formal parameter.

```
with <vendor_name>_System_Package;
with Parts_Domains; use Parts_Domains;
package Parts_Module is

    procedure Delete_Parts (
        Part_Name              : Pname_Type;
        Transaction_Identifier :
            <vendor_name>_System_Package.Transaction_Identifier_Type);

end Parts_Module;
```

Alternatively, the transaction identifier may explicitly appear as a parameter to the SAMeDL procedure. In this case the transaction identifier may or may not appear as part of the SAMeDL statement in the procedure, but the compiler generates an Ada procedure at the abstract interface that includes a transaction identifier formal parameter.[28] The transaction identifier formal parameter must have a domain reference. The domain can be supplied in a predefined SAMeDL definitional module supplied by the vendor, such as <*vendor_name*>_Domains below.[29]

```
definition module <vendor_name>_Domains is

    domain Transaction_Domain is implementation-defined;

end <vendor_name>_Domains;


with Parts_Domains;  use Parts_Domains;
with <vendor_name>_Domains;
abstract module Parts_Module is
    authorization Parts_Suppliers_Database

    procedure Delete_Parts (
        Input_Pname named Part_Name : Pname;
        Transaction_Identifier          :
            <vendor_name>_Domains.Transaction_Domain) is

        delete from P
            where PNAME = Input_Pname;

end Parts_Module;
```

In the above example, the *Transaction_Identifier* formal parameter to the SAMeDL procedure is not used within the SAMeDL statement. The SAMeDL compiler produces the following Ada code in this case.

```
with <vendor_name>_Domains;
```

[28] The case where the transaction identifier is supplied as a parameter to the SAMeDL procedure, yet does not appear in the SAMeDL statement, is an extension of the semantics but not the syntax of the SAMeDL. Here the purpose of the parameter is to signal the compiler to produce a formal transaction identifier parameter in the associated Ada procedure, rather than to supply a parameter to the SAMeDL statement.

[29] Of the alternatives presented, this is the only one requiring a transaction identifier domain.

```
with Parts_Domains; use Parts_Domains;
package Parts_Module is

    procedure Delete_Parts (
        Part_Name              : Pname_Type;
        Transaction_Identifier :
            <vendor_name>_Domains.Transaction_Domain_Type);

end Parts_Module;
```

### 4.4.3. An Example Abstract Module

This section presents an Ada code fragment as an example of the procedure bodies in the Ada package body of an abstract module.[30] The code fragment example illustrates the procedure bodies for the *Delete_Parts*, *Connect*, and *Disconnect* procedures as they appear in the SAMeDL abstract module *Parts_Module* below. The code fragment is an example of the code that might be produced by a SAMeDL compiler targeted for the SYBASE DBMS.[31]

For the following SAMeDL abstract module,

```
with Parts_Domains; use Parts_Domains;
with Database_Domains; use Database_Domains;
extended abstract module Parts_Module is
    authorization Parts_Suppliers_Database

    extended procedure Connect (Database : Database_Name) is
       connect Database;

    procedure Delete_Parts (
        Input_Pname named Part_Name : Pname) is

        delete from P
            where PNAME = Input_Pname;

    extended procedure Disconnect (Database : Database_Name) is
        disconnect Database;

end Parts_Module;
```

assume that the definitional modules *Database_Domains* and *Parts_Domains* are declared as follows.

```
with SAMeDL_Standard;
definition module Database_Domains is

    domain Database_Name is new SAMeDL_Standard.SQL_Char(
        Length => 24);
```

[30]The author makes no claims that the code presented in these examples will compile, or that they represent the proper use of SYBASE Ada/DB-Library routines.

[31]SYBASE here refers to Release 4.0 of the SYBASE DBMS for VMS.

```
end Database_Domains;

with SAMeDL_Standard;
definition module Parts_Domains is

    domain Pname is new SAMeDL_Standard.SQL_Char(
        Length => 20);

end Parts_Domains;
```

The abstract interface for abstract module *Parts_Module* looks like this.

```
with SYBASE_TYPES;              -- Includes all SYBASE type definitions
with SYBASE_INTERFACE;          -- Includes all SYBASE Ada/DB-Library
with SYBASE_System_Package;     --    function and procedure declarations
with Database_Domains; use Database_Domains;
with Parts_Domains; use Parts_Domains;
package Parts_Module is

    procedure Connect (Database               : Database_Name_Type;
                       Transaction_Identifier :
                           out SYBASE_TYPES.DBPROCESS_PTR);

    procedure Delete_Parts (
        Part_Name              : Pname_Type;
        Transaction_Identifier : SYBASE_TYPES.DBPROCESS_PTR :=
            SYBASE_System_Package.Default_Transaction_Identifier);

    procedure Disconnect (Transaction_Identifier :
                              SYBASE_TYPES.DBPROCESS_PTR);

end Parts_Module;
```

Notice that the Transaction_Identifier_Type is the SYBASE DBPROCESS_PTR type. An object of this type is a pointer to a communication area for a particular user connection to a database. Notice also that the package SYBASE_System_Package is a new package, not currently supplied by SYBASE, containing the Default_Transaction_Identifier for use by an application program performing only sequential transaction processing. An example of this package appears below.

```
with SYBASE_TYPES;
package SYBASE_System_Package is

    Default_Transaction_Identifier : SYBASE_TYPES.DBPROCESS_PTR;

end SYBASE_System_Package;
```

Below is the Parts_Module Ada package body corresponding to the Ada package specification above.

```
with SQL_Char_Pkg; use SQL_Char_Pkg;
with SQL_Communications_Pkg; use SQL_Communications_Pkg;
with SQL_Database_Error_Pkg; use SQL_Database_Error_Pkg;
package body Parts_Module is
```

```
-- Connect procedure.

procedure Connect (Database               : Database_Name_Type;
                  Transaction_Identifier :
                      out SYBASE_TYPES.DBPROCESS_PTR) is

    TRANS_ID     : SYBASE_TYPES.DBPROCESS_PTR;
    LOGIN_RECORD : SYBASE_TYPES.LOGINREC_PTR;
    Null_Server  : String(1..0) := "";

begin

    -- Get a login record.
    LOGIN_RECORD := SYBASE_INTERFACE.ADBLOGIN;

    -- Connect to the database.
    if Is_Null(Database) then

        -- Connect to the default database.
        TRANS_ID := SYBASE_INTERFACE.ADBOPEN(LOGIN_RECORD,
                                             Null_Server);

    else

        -- Connect to the named database.
        TRANS_ID := SYBASE_INTERFACE.ADBOPEN(LOGIN_RECORD,
                                             To_Unpadded_String(
                                             Database));

    end if;

    -- Check that the connection was successful.
    -- The following is the default status processing for the SYBASE
    --    calls to connect to a database (using ADBOPEN).
    -- If the TRANS_ID pointer is null, then the connection failed.
    -- If the TRANS_ID pointer is not null, then the connection was
    --    successful, and the pointer points to a communication area
    --    for the user connection.
    if TRANS_ID = null then

        -- Raise SQL_Database_Error exception.
        Process_Database_Error;
        raise SQL_Database_Error;

    end if;

    Transaction_Identifier := TRANS_ID;

end Connect;

-- Delete procedure.

procedure Delete_Parts (
    Part_Name              : Pname_Type;
    Transaction_Identifier : SYBASE_TYPES.DBPROCESS_PTR :=
        SYBASE_System_Package.Default_Transaction_Identifier) is
```

```
            Return_Code : SYBASE_TYPES.RETCODE;

        begin

            -- First put the command into the command buffer.
            if Is_Null(Part_Name) then

               SYBASE_INTERFACE.ADBCMD (Transaction_Identifier,
                                         "delete from P where PNAME is null");

            else

               SYBASE_INTERFACE.ADBCMD (Transaction_Identifier,
                                         "delete from P where PNAME = " &
                                         To_Unpadded_String(Part_Name));

            end if;

            -- Next, execute the command.
            Return_Code := SYBASE_INTERFACE.ADBSQLEXEC (Transaction_Identifier);

            -- Finally, check the return code.
            -- The following is the default status processing for the SYBASE
            --    call to execute an SQL statement (using ADBSQLEXEC).
            -- If the Return_Code variable has the value SYBASE_TYPES.SUCCEED,
            --    then the statement executed successfully.
            -- If the Return_Code variable has the value SYBASE_TYPES.FAIL,
            --    then the statement did not execute successfully.
            if Return_Code /= SYBASE_TYPES.SUCCEED then

                -- Raise SQL_Database_Error exception.
                Process_Database_Error;
                raise SQL_Database_Error;

            end if;

        end Delete_Parts;

        -- Disconnect procedure.

        procedure Disconnect (Transaction_Identifier :
                                   SYBASE_TYPES.DBPROCESS_PTR) is

        begin

            -- Disconnect from the database.
            SYBASE_INTERFACE.ADBCLOSE (Transaction_Identifier);

            -- There is no status variable to check.

        end Disconnect;

    end Parts_Module;
```

**Note**: SYBASE does not implement the ANSI standard SQLCODE return code mechanism. As can be seen from the example above, SYBASE has a RETCODE data type, an enumeration type that is used to determine the success or failure of the Ada DB-Library procedure call.

## 4.5. Application Design Issues

There are three issues that an Ada application programmer must be aware of when constructing an application that processes multiple concurrent transactions:

1. Maintenance of transaction identifiers.
2. Synchronization of database operations within individual transactions.
3. Communication between transactions.

The first of these is a natural side effect of having more than one transaction in a single application. This is described in greater detail in Section 4.5.1 below.

The last two issues arise because Ada tasking conflicts with the traditional DBMS model of transaction processing. In the traditional model of transaction processing, concurrent transactions in a multi-user DBMS environment are considered to be independent and non-interfering. Transactions executing concurrently are guaranteed by the ANSI standard to be serializable (see Section 4.16 of [4]). That is, the execution of concurrent transactions produces the same effect as if the transactions were executed serially, where one transaction runs to completion before another one is begun. Ada tasking, however, has the potential to destroy this model of transaction processing. This will be discussed in greater detail in Sections 4.5.2 and 4.5.3 below.

Each of these issues is discussed here because this is the first time that they appear in the process of developing a DBMS application program. Ada tasking has a tremendous impact on the design of such an application by introducing concurrency in a domain designed for sequential processing.

### 4.5.1. Maintenance of Transaction Identifiers

Section 4.2.2 above discusses the necessity for transaction identifiers in an application program that processes multiple concurrent transactions.

It is the responsibility of the application programmer to declare and maintain all transaction identifier objects for the multiple transactions that will be implemented by the application program. The application programmer must *with* the Ada package supplied by the SAMeDL compiler vendor that contains the Ada type declaration for transaction identifier objects (see Section 4.4.1 above) for visibility to this data type. The transaction identifier objects are then declared within the Ada application program.

It is also the responsibility of the application programmer to maintain the mapping of database operations to the multiple transactions within an application using transaction identifiers. Multiple concurrent transactions, however, present a new class of problems to the application designers and programmers. If the transaction identifiers are not managed properly, the potential exists for database operations to be unintentionally designated for the wrong transactions. If an error such as this occurs, chances are that the application will still run to completion, and the error may go undetected. The data in the DBMS may be corrupted in this case. Errors such as these are difficult to find, and can be very destructive.

### 4.5.2. Synchronization Within Individual Transactions

As described above, a transaction is comprised of a sequence of database operations. In traditional DBMS application programs developed in programming languages other than Ada, only one thread of execution is possible. This single thread of execution guaranteed that the execution of the database operations comprising a transaction would remain serial.

In a DBMS application program written in Ada, however, the database operations that comprise a single transaction can be distributed among more than one task. If transactions in an Ada program are mapped to tasks in this way, the sequential nature of the transaction can be destroyed. In this case, the potential exists for more than one database operation to execute simultaneously on behalf of a single transaction. Commercial DBMS cannot tolerate this, as they support only serial execution of database operations within a single transaction. If an application program must map individual transactions to many Ada tasks in this way, then the many tasks associated with a single transaction must all use a synchronization task to guarantee serial execution of the database operations.[32]

The synchronization task for this type of application can be designed in one of two ways. First, it can be designed to contain the database services for all the tasks associated with the synchronization task's transaction. This guarantees serial execution of all the database services of a particular transaction, because the synchronization task can execute only one service at a time. This design may not be desirable in that it may require a large number of task entries for the synchronization task.

In the second case, invocations of the database services are distributed among the tasks comprising the transaction. The synchronization task in this case provides a semaphore service. The application is responsible for acquiring the semaphore prior to calling a database service of a particular task, and then releasing it upon return from the call.

### 4.5.3. Communication Between Transactions

In the traditional model of transaction processing, transactions are considered to be non-interfering and execute without intercommunication. In an Ada DBMS application program, however, two transactions executing in independent tasks may communicate with each other through global variables and rendezvous.

---

[32] See Section 9.2 of [11] for a complete discussion of SQL and Ada tasks.

This communication can result in the sharing of information derived from the execution of the database operations within the transactions themselves. Furthermore, the execution of subsequent database operations in a transaction may be determined by the information they share. In this case, the transactions are no longer considered to be non-interfering, and are not serializable. This conflicts with the traditional model of transaction processing implemented by commercial DBMS. In fact, the DBMS is unaware that such a communication exists, interpreting the individual transactions as serial.

This situation may not be erroneous. The success of a particular application may depend upon such semantics. Circumventing the long-established transaction processing paradigm of non-interference, however, should not be attempted frivolously. Such applications must be designed carefully to avoid the many potential problems that could arise.

# 5. SQL Decimal Support in the SAMeDL

**by Gregory Zelesnik**

## 5.1. Introduction

To be accepted as a viable alternative to COBOL in the world of information management, Ada must be able to provide at least the same level of support for decimal data semantics. Current implementations of American National Standards Institute (ANSI) standard Ada compilers, however, do not provide this type of support, which is contributing to the slower acceptance of Ada for Management Information Systems (MIS) applications.

It has been suggested that if Ada were changed to provide a new pragma, a pragma *Packed_Decimal* for instance, for fixed-point types so that the underlying concrete representation,[33] were packed decimal,[34] then Ada fixed-point types could be used to provide the same decimal data semantics that COBOL provides with the *Computational-3* data type. This is not quite true, however, for the following reason.

As discussed in [2], there is a strong assumption that the underlying implementation of fixed point in an Ada compiler will be in binary, as discussed in 3.5.9 of [3], which defines the model numbers as covering a binary range, described by the mantissa value B. Current Ada compiler implementations, therefore, require that the specified value for the *'SMALL* attribute (the *small* value) of a fixed point type be a power of two. This means that decimal data cannot be represented exactly by Ada fixed point types, since this would necessitate the ability to specify decimal *small* values. Therefore, regardless of whether the underlying representation for fixed point quantities is packed decimal, or integer, Ada fixed point types cannot provide decimal data semantics.[35]

Because Ada fixed-point types cannot be used, it is necessary to try to implement decimal data semantics in Ada by using the abstract typing facilities available in the language. The following sections discuss alternatives for SQL decimal support in the SAMeDL. This discussion centers mainly on providing support for underlying packed decimal and integer representations, but refers occasionally to other types of support.

---

[33] Concrete representation refers to the organization and the number of bits that describe the data.

[34] The packed decimal representation is a scaled integer representation in which one digit is stored in four bits, with two digits to a byte. The number is followed by a sign, also represented in four bits, and zero filled on the left when there is an even number of digits.

[35] It is important to note (see [2]) that there is no relation between packed decimal and decimal *small* values. Packed decimal is simply another way of representing integers, and it is possible to use packed decimal for any fixed-point values, including those with binary *small* values.

Support for SQL decimal data in the SAMeDL must be provided at two different levels. First, a binding between an Ada data type and an SQL data type must be selected for the particular implementation of SQL of the DBMS. This binding facilitates the transfer of the decimal data across the concrete interface (see Chapter 1 of [11]) into the abstract module without conversion. This level of support will be referred to throughout the chapter as the concrete level.

Second, support must be provided on a more abstract level. A base domain support package must be created to implement data semantics for decimal data in the Ada application, and a base domain must be declared in the SAMeDL to provide the SAMeDL compiler with the information necessary to generate decimal domains and Ada and SQL interface code. This level will be referred to as the application level.

## 5.2. Concrete Level

Support at the concrete level consists of an Ada/SQL binding[36] for decimal data at the concrete interface. The objective is to choose data types in both Ada and SQL that have identical concrete representations so that no data conversion is necessary as the data pass across the interface. Section 8 of [5] specifies the ANSI standard programming language type bindings for Ada and SQL, but adherence to the standard may be too restrictive because it prohibits the use of the SQL data type *decimal* in a parameter declaration. The following two sections discuss Ada/SQL bindings that may be of interest to the SAMeDL compiler implementor. Both ANSI standard bindings and bindings not described by this standard are presented.

### 5.2.1. Standard Bindings

The embedded SQL standard (see [5]) requires that the SQL data type in an Ada/SQL binding specify either *character*, *smallint*, *integer*, *real*, or *double precision*. The Ada type of this binding is required to be a data type from the Ada package *SQL_Standard*.[37] The standard also stipulates that, for the SQL data types listed above, their Ada counterparts in the binding *must* be the Ada types *SQL_Standard.Char*, *SQL_Standard.Smallint*, *SQL_Standard.Int*, *SQL_Standard.Real*, and *SQL_Standard.Double_Precision*, respectively.

For the SAMeDL compiler implementor supporting *decimal* data, the bindings of interest are the numeric ones. These can be further partitioned into two classes: integer bindings and floating-point bindings. The integer bindings include the *SQL_Standard.Int*/*integer* binding and the *SQL_Standard.Smallint*/*smallint* binding. The floating point bindings include *SQL_Standard.Real*/*real* and *SQL_Standard.Double_Precision*/*double precision*. The next two sections discuss these two classes of bindings in greater detail.

---

[36]An Ada/SQL binding is the specification of an Ada data type/SQL data type pair that is used to describe a parameter at the concrete interface between the Ada procedure and the corresponding SQL procedure.

[37]For a more detailed description of the Ada package *SQL_Standard*, please refer to [11].

Notice that **decimal** is excluded from the list of SQL data types that have standard bindings (see Section 8 of [ansi2]). This means that decimal data type must be converted to another data type in the SQL module before it can be passed across the concrete interface, if the Ada/SQL bindings are to conform to the standard. This requirement for conversion is a major disadvantage of using a standard Ada/SQL binding, because the potential exists for significant losses in precision,[38] scale, and accuracy in the data.

#### 5.2.1.1. Integer Bindings

An integer binding converts decimal data to binary integer data in the SQL module. There are two issues that must be addressed for integer bindings. First, the conversion process may cause a loss of the fractional portion of the decimal data. Second, because the *decimal* data type has the potential to represent data with more significant digits than any integer data type, the conversion from decimal to integer may cause an overflow exception in the DBMS for extremely large values.

SAMeDL programmers can overcome the first of these deficiencies by scaling the decimal data in the SAMeDL module so that the fractional data are not lost when the conversion to integer is performed. This approach, however, also requires that the database administrator provide appropriate support at the application level (see Chapter 5.3) to implement decimal data semantics using this scaled integer data.

The second problem can be avoided by placing constraints on the range of decimal values that can be stored in the DBMS. This range constraint must account for significant digits, meaning that the number of scale digits plus integral digits for every value in the range must never exceed the number of significant digits available for the integer data type in the Ada/SQL binding. Furthermore, each decimal value in the range must never be greater than the largest integer value available for the integer type in the binding *after* scaling has been performed. If every decimal value in the DBMS conforms to these range constraints then there should never be an overflow exception raised during the conversion process. It is often very difficult, however, to enforce these range constraints. They must be applied to the decimal columns consistently in every application program using the data, and must also be applied to the columns themselves in the DBMS to prevent violations due to ad hoc query updates.

An advantage of using an integer binding is that, if scaled properly, the data will retain exactness. No loss of accuracy will occur during the conversion.

---

[38] The use of the term *precision* in this chapter is consistent with its use in [4], where it is used to refer to the number of significant digits.

#### 5.2.1.2. Floating-Point Bindings

A floating-point binding converts the decimal data to an SQL real representation in the SQL module. This conversion may cause a loss of accuracy in the data because the values are transformed from their exact representation into an approximate one. Furthermore, because the *decimal* data type has the potential to represent data with more significant digits than any real data type, the conversion from decimal to real may cause a significant loss in the precision of the decimal data. An advantage of using a floating-point binding, however, is that no scaling of the data is necessary in the SQL module and the application.

### 5.2.2. Non-Standard Bindings

Non-standard Ada/SQL bindings are those bindings not described by the ANSI SQL standard (see Section 8 of [5]). As in the case of *decimal*, described below, the SAMeDL compiler implementor may be required to bind SQL data types to Ada data types that *do not* reside in the Ada package SQL_Standard. In this case, the most important aspect is that the Ada type and the SQL type have identical underlying concrete representations so that the data are not converted as they are passed across the concrete interface. The difficulty with binding the SQL data type *decimal* to an Ada type is that there is no predefined data type in Ada that has an identical concrete representation and data semantics. One must be created by the SAMeDL compiler implementor and made available to the database administrator so that support for decimal at the application level (see Chapter 5.3) can be implemented. The next section describes one particular example of an Ada type that can be bound to the SQL *decimal* data type. It should not be inferred that this particular example is the one and only way to construct the Ada data type in a decimal binding. In fact, there are many ways that such a type can be constructed. This discussion is merely intended to introduce the issues that the SAMeDL compiler implementor must address when constructing the type.

### A Non-Standard Binding for Decimal

There are two characteristics of decimal data that influence the construction of the Ada type for the decimal binding. These characteristics are:

1. The underlying concrete representation of the decimal data.
2. The precision of the SQL module decimal parameter.

Both of these are addressed in the following paragraphs as the example Ada type is discussed in greater detail. The name of the Ada data type in the example is Max_Decimal,[39] and it is constructed for use in a binding with SQL decimal data whose underlying concrete representation is packed decimal.

The first characteristic that must be examined is the concrete representation of the decimal data. Max_Decimal is defined to be a bit receptacle for packed decimal data. It is con-

[39]The Ada package Concrete_Decimal, shown in Appendix B, contains the full specification of the concrete type Max_Decimal.

structed as an array type since no predefined Ada scalar type can accommodate the large decimal values. The component of the array accommodates one byte, which is the level of granularity required for the packed decimal operations on the underlying hardware. It does not matter which scalar type is chosen for the array component, so long as it requires only one byte for storage. For Max_Decimal, the component type is an eight-bit integer type called *Digit*, although an eight-bit character type was also considered. To guarantee that only eight bits are used by the Ada compiler to represent objects of type Digit, a representation clause is used.

```
-- an object of this type can hold two decimal digits,
--    or one digit and the sign
type Digit is range -(2**7)..(2**7)-1;

-- the representation clause below guarantees an array
--    component of eight bits
for Digit'size use 8;
```

The second characteristic that influences the construction of the Ada type is the precision of the SQL module decimal parameter. This precision defines the number of decimal digits a number can have and effectively defines the requirements for the size of the Ada array type.

There are many approaches that can be taken when constructing the Ada type to account for precision. One approach is to create a different Ada array type for every possible precision allowable in the DBMS. Another approach is to create an array type to accommodate the maximum allowable precision in the DBMS, and encapsulate this type in a record type with another component that stores a value for the actual precision. A third approach is to create an array type to accommodate the maximum allowable precision in the DBMS, and enforce the specified precision using range constraints for values of the type.[40] Max_Decimal is constructed using the last of these approaches. The maximum precision allowable for objects of this type is determined by the constant MAX_SIZE. The array type, therefore, is constrained to have 16 positions to account for all of the digits plus the sign of the decimal number as well.

The Max_Decimal data type is declared as follows.

```
-- the maximum number of digits of precision for the
--    decimal data
MAX_DIGITS : constant Integer := 31;

-- this constant determines the number of bytes needed
--    by Max_Decimal to store MAX_DIGITS number of
--    decimal digits plus the sign
MAX_SIZE : constant Integer := ((4 * (MAX_DIGITS))/Digit'Size) +1;

-- can store MAX_DIGITS number of decimal digits
--    plus the sign
type Max_Decimal is array (1..MAX_SIZE) of Digit;
```

[40] See Section 5.3.1.2 for more information on the implementation of range constraints for abstract types based on this concrete Ada type.

The Max_Decimal data type is declared to be *private* in package Concrete_Decimal (see Appendix B) so that the contents of the array type cannot be directly manipulated. It is not declared to be *limited* because Ada predefined assignment and equality are correct in this case. No arithmetic operators are defined for Max_Decimal. These operations are implemented for the abstract data types based on Max_Decimal at the application level discussed in Chapter 5.3.

Max_Decimal can now be used in a decimal binding. Because the type can accommodate the maximum precision allowable in the DBMS, namely 31 digits, the SQL decimal data type in the binding must also specify 31 digits of precision. The Ada/SQL binding for this example, therefore, is the *Concrete_Decimal.Max_Decimal/decimal(31,n)* data type pair, where *n* represents the specified scale of the decimal SQL module parameter. The scale of the decimal parameter does not affect the construction of the Ada type in the binding, but does affect the construction of the abstract data types in the application level. Scale is discussed in much greater detail in Section 5.3.

### Implementation Note

The SAMeDL compiler implementor must recognize that, for non-standard Ada/SQL bindings, the implementation of decimal parameter passing across the concrete interface may become non-portable as the Ada data type in the binding becomes more complex.

For more complex Ada data types, the implementation of the *pragma INTERFACE* parameter passing techniques become more complex, and thus may become non-portable, even between subsequent versions of the same Ada compiler. Therefore, the SAMeDL compiler implementor must weigh the complexity of the underlying parameter passing techniques when constructing the Ada type in a decimal binding, and choose the Ada data type wisely.

## 5.2.3. Summary

In summary, whether or not the Ada/SQL binding for decimal data is an ANSI standard one, the primary objective for choosing a binding is to select an Ada data type and an SQL data type that have identical concrete representations so that the decimal data are not converted as they are passed across the concrete interface. If the Ada data type in the binding is not one of the standard types, the SAMeDL compiler implementor must weigh the benefits of the complexity of the data type against the potential non-portability of the binding.

## 5.3. Application Level

Support at the application level consists of:

- Creating a base domain support package to implement decimal data semantics.
- Declaring the associated base domain in the SAMeDL.

The first of these two tasks can be accomplished by an Ada programmer who is familiar with decimal data semantics. The second can be accomplished by a SAMeDL programmer. The rest of this chapter discusses the steps necessary to complete the application level of support for a *decimal* base domain in the SAMeDL. The focus of the discussion is a complete example of application support based on the non-standard binding described in Section 5.2.2; however, support based on the standard integer binding discussed in Section 5.2.1.1 will be outlined as well.

### 5.3.1. Base Domain Support Package

Support at the application level begins with the creation of the Ada support package for the base domain. This package contains abstract type definitions built upon the Ada type of the Ada/SQL binding, and procedures, functions, and subpackages necessary to fully implement the decimal data semantics of the DBMS data in the Ada application. The next two sections discuss the creation of the abstract types and operations on those types for a *decimal* base domain called SQL_Decimal.

#### 5.3.1.1. Abstract Data Types

The data semantics for decimal data in the Ada application are implemented by the creation of two abstract data types and a set of operations on each type. The first type, called the not null-bearing type, is constructed to implement decimal data semantics in Ada. An object of this type can never have the SQL *null* value. The second type, called the null-bearing type, is constructed from the not null-bearing type, and implements the data semantics of the SQL *null* value in addition to the decimal semantics already defined for the not null-bearing type.[41] An object of this type may contain the SQL *null* value. The next two sections describe each of these two data types in detail, and present examples for the SQL_Decimal base domain.

**Not Null-Bearing Type**

The not null-bearing type is usually derived or constructed from the concrete type.[42] It is the set of operations for the not null-bearing type (*not* the concrete type) that implements the data semantics for the DBMS data. The concrete type is simply the Ada description of the SQL module parameter at the concrete interface.

[41] If the DBMS of choice does not support null values, the Ada programmer should still consider implementing a null-bearing type in the base domain package for portability between DBMS.

[42] Concrete type refers to the Ada data type of the Ada/SQL binding.

The type definition for Max_Decimal, the concrete type for the decimal example described in Section 5.2, is not sufficient for fully implementing decimal data semantics. Therefore, the not null-bearing type definition for the SQL_Decimal base domain must be constructed from Max_Decimal and extended to incorporate the missing information.

The one characteristic of decimal data that is not reflected in the type definition for Max_Decimal is scale. Every decimal value has a scale associated with it, even if the scale is zero. Scale information must be known for the operands in decimal arithmetic operations for the operations to produce accurate results. This information must be reflected in each decimal object, and therefore must be incorporated into the not null-bearing type.

There are several approaches for incorporating scale information into the not null-bearing type of the SQL_Decimal base domain, SQL_Decimal_Not_Null. One is to declare a not null-bearing type for each scale possible in the DBMS. Using the non-standard concrete type for packed decimal described in Section 5.2.2, the type declarations are as follows.

```
-- can store MAX_DIGITS number of decimal digits
--    plus the sign
type SQL_Decimal_Not_Null_1 is new Max_Decimal;
type SQL_Decimal_Not_Null_2 is new Max_Decimal;
type SQL_Decimal_Not_Null_3 is new Max_Decimal;

                              .
                              .
                              .

type SQL_Decimal_Not_Null_31 is new Max_Decimal;
```

This approach, however, causes an explosion of overloaded procedures and functions when decimal data semantics are implemented for each one of the types, and therefore it may not be considered a viable one.

A second approach is to encapsulate the scale information with the decimal value in a record type. This encapsulation may take the form of a simple record type, with scale as another component,

```
-- range of allowable scale in the DBMS
subtype Scale_Digits is Natural range 0 ..  MAX_DIGITS;

-- scale as a record component
type SQL_Decimal_Not_Null is record
    Scale : Scale_Digits;
    Value : Max_Decimal;
end record;
```

or it may take the form of a discriminated record type, with scale as the discriminant.

```
-- range of allowable scale in the DBMS
subtype Scale_Digits is Natural range 0 ..  MAX_DIGITS;

-- scale as a discriminant
type SQL_Decimal_Not_Null (Scale : Scale_Digits) is record
    Value : Max_Decimal;
end record;
```

The discriminated record approach is preferable if the record type is declared to be private, because then the scale can be set when an object of the type is declared without requiring the use of an initialization procedure or function. The discriminated record type declaration above is similar to the one found in the example of the SQL_Decimal base domain support package (see Appendix C).

To illustrate the construction of a not null-bearing type based on a concrete type for a standard Ada/SQL binding, the following record declaration for SQL_Decimal_Not_Null is shown to contain a *value* component of type SQL_Standard.Int. This *value* component is used to store scaled integer data from the DBMS of the type discussed in Section 5.2.1.1 above. In fact, the data semantics for this representation will be identical to those of the above declaration for the packed decimal representation.

```
-- range of allowable scale in the DBMS
subtype Scale_Digits is Natural range 0 ..  MAX_DIGITS;

-- scale as a discriminant
type SQL_Decimal_Not_Null (Scale : Scale_Digits) is record
    Value : SQL_Standard.Int;
end record;
```

Once the not null-bearing type has been constructed, operations for this type must be implemented to provide decimal data semantics in the Ada application program. These operations are encapsulated with the not null-bearing type in the base domain support package. The decimal operations are described more fully in Section 5.3.1.2.

**Null-Bearing Type**

The null-bearing type of a base domain is used to implement the SQL semantics of the *null* value in the Ada application, and is built using the not null-bearing type. There are many ways to use the abstract typing facilities of Ada to encapsulate the notion of nullness with an object of the not null-bearing type. Usually, however, the null-bearing type is constructed as a record type with a not null-bearing component and a boolean component used to indicate whether or not the object is null.

Two approaches for constructing the null-bearing type, called SQL_Decimal, are discussed here. The first approach is to construct SQL_Decimal as a discriminated record type, where one of the discriminants is a boolean component used to indicate the nullness of the object.[43]

```
-- "Is_Null" as a discriminant
type SQL_Decimal (Is_Null : Boolean;
                  Scale   : Scale_Digits) is record
    Value : SQL_Decimal_Not_Null(Scale);
end record;
```

The second approach is to construct a similar discriminated record type, but the boolean component indicating nullness is a non-discriminant component.

```
-- "Is_Null" as a record component
type SQL_Decimal (Scale : Scale_Digits) is record
    Is_Null : Boolean;
    Value   : SQL_Decimal_Not_Null(Scale);
end record;
```

The latter of the two approaches was implemented in the base domain support package for SQL_Decimal (see Appendix C).

Once the null-bearing type has been constructed, operations for this type must be implemented to provide SQL data semantics in the Ada application program. These operations are discussed in Section 5.3.1.2.

**Need for Encapsulation**

Both the not null-bearing and the null-bearing types of the SQL_Decimal base domain are declared to be limited private (see Appendix C).

For objects of the SQL_Decimal_Not_Null type, the *value* component is meaningless when referenced separately from the rest of the record type. Without the scale, the value cannot be interpreted correctly. For this reason the application programmer must be prevented from manipulating the record components directly. Preventing direct manipulation is accomplished by declaring SQL_Decimal_Not_Null to be private, and providing operations on the type in SQL_Decimal_Pkg (see Appendix C) that allow the application programmer to manipulate objects of this type. SQL_Decimal_Not_Null is further declared to be limited because Ada predefined equality is not desirable,[44] and assignment is redefined to implement range constraints for values of the type (see Section 5.3.1.2 for further details).

SQL_Decimal is declared to be private for essentially the same reason. The type is also limited because it has a limited component.

[43]Note that at least one discriminant for SQL_Decimal is necessary, and it must represent scale information. This discriminant is necessary because it must be supplied to the *value* component that is declared to be of the SQL_Decimal_Not_Null type. This type requires a value for *its* discriminant representing scale.

[44]Two SQL_Decimal_Not_Null objects with equal values but different scales are not bitwise identical.

### 5.3.1.2. Operations

Operations are implemented on the abstract types of the base domain to provide the decimal data semantics of the DBMS data in the Ada application. Operations on the not null-bearing type provide the decimal data semantics, and operations on the null-bearing type provide the additional data semantics of the SQL *null* value.

The following sections discuss the important issues with respect to providing operations for decimal objects of the abstract types discussed in Section 5.3.1.1 above. The operations presented are the ones implemented for SQL_Decimal in the Ada package SQL_Decimal_Pkg (see Appendix C).

**Not Null-Bearing Operations**

The not null-bearing type of the SQL_Decimal base domain (see Appendix C) is a limited private type for the reasons discussed in Section 5.3.1.1 above. This means that all of the operations that provide data semantics for this type must be implemented. For SQL_Decimal_Not_Null, these operations implement decimal data semantics in Ada for data that have an underlying packed decimal concrete representation. The implemented data semantics include decimal arithmetic, conversion, and attribute operations.

**Decimal Arithmetic**

To implement decimal arithmetic in Ada for the not null-bearing type, comparison operators and unary and binary arithmetic operators must be implemented in the base domain support package. Comparison operators =, /=, >, <, >=, and <= are implemented that compare two operands of the not null-bearing type and return the boolean values *true* and *false*. The unary operators +, -, and *abs* are implemented to operate on a single object of the not null-bearing type, returning that object with the original value, a negated value, and a positive value, respectively. Finally, the binary +, -, ∗, and / operators are implemented to provide addition, subtraction, multiplication, and division operations for two decimal operands of the not null-bearing type.

Each of these operations is implemented for SQL_Decimal_Not_Null in the Ada package SQL_Decimal_Pkg as a function. Since SQL_Decimal_Not_Null contains a packed decimal representation of the decimal value, each of these function bodies calls an assembler subroutine to make use of packed decimal arithmetic CPU instructions available on the underlying hardware platform to perform the operations.[45] The result is then returned to the function. Additionally, since the assembler subroutines for the binary operations essentially only perform integer arithmetic on packed decimal values, the scale of the result is calculated

[45] The semantics of the operations on SQL_Decimal_Not_Null objects are given by the underlying instruction set architecture of the machine.

and set by each of these functions.[46]

If SQL_Decimal_Not_Null contained an integer concrete representation of the decimal value (see Section 5.3.1.1) instead, the bodies for each of the functions mentioned above would perform their operations using Ada predefined integer operators on the integer values to simulate decimal arithmetic. Again, since the binary operations are essentially performing integer arithmetic, the scale of the result must be calculated and set by the functions for these operations.

**Conversion**

Conversion operations are provided for SQL_Decimal_Not_Null in the Ada package SQL_Decimal_Pkg for Ada application programmer convenience. There are four functions that convert between SQL_Decimal_Not_Null and the not null-bearing types of SAMeDL standard numeric base domains SQL_Int and SQL_Double_Precision. Two functions are also provided that convert between SQL_Decimal_Not_Null and the SAMeDL standard character base domain SQL_Char. Finally, a function that converts an object from SQL_Decimal_Not_Null to the Ada predefined type Standard.String is also provided.

The Ada Constraint_Error exception is raised in each of the functions that convert *to* the SQL_Decimal_Not_Null type if the value requires more significant digits than the maximum decimal precision allowed (specified in the constant MAX_DIGITS).

The Ada Constraint_Error exception is also raised in the conversion function To_SQL_Int_Not_Null if the decimal value is too large to be represented by an object of the SQL_Int_Not_Null data type.

**Attributes**

Functions that simulate Ada attribute functions are also provided for SQL_Decimal_Not_Null in package SQL_Decimal_Pkg for convenience. These functions accept an operand of the not null-bearing type, and return attribute information about the operand itself. The *Width* function returns the length of the Standard.String representation of the decimal value. The *Integral_Digits* function returns the maximum number of digits a decimal value can have to the left of the decimal point for an object of the type. The *Scale* function returns the maximum number of digits a decimal value can have to the right of the decimal point for an object of the type. The *Fore* function returns the actual number of digits to the left of the decimal point that an object of the type has. The *Aft* function returns the actual number of digits to the right of the decimal point that an object of the type has.

The *Machine_Rounds* and *Machine_Overflows* functions accept an operand of the not null-bearing type, and return an Ada Standard.Boolean value. They simulate the Ada attribute

[46] For example, the scale of the result of the addition of two objects whose values are 1.2 and 2.45, respectively, is 2, since the result is 3.65.

functions that have the same name. *Machine_Rounds* returns *true* if the result returned by each of the binary operations is rounded if it cannot be exact. *Machine_Overflows* returns *true* if the Constraint_Error exception is raised in overflow situations.

**Null-Bearing Operations**

The operations for the null-bearing type build upon the operations for the not null-bearing type, adding data semantics for the SQL *null* value. In addition to the decimal arithmetic, conversion, and attribute operations described above for the not null-bearing type, the null-bearing type has an operation that returns an object of the null-bearing type whose value is null, and testing operations that indicate whether or not an object has the null value.

**Decimal Arithmetic**

The same arithmetic and comparison operators implemented for objects of the not null-bearing type are also implemented for the null-bearing type. They differ, however, in the following ways. Any arithmetic operation applied to an object whose value is null results in the null value. Otherwise the arithmetic operation is identical to the operation for the not null-bearing type. There are two sets of comparison operators for the null-bearing type. One set returns *Boolean_With_Unknown* values (see Section 3.1.2 of [11]), and the other returns Standard.Boolean values. For the Boolean_With_Unknown comparison operators, the comparison of any value to an object whose value is *null* results in a new truth value called *unknown*. Otherwise the comparison operation is identical to the operation for the not null-bearing type. For the Standard.Boolean comparison operators, the comparison of any value to an object whose value is *null* results in the truth value *false*. Otherwise the comparison operation is identical to the operation for the not null-bearing type.

**Conversion**

Conversion operations are provided for the null-bearing type in the Ada package SQL_Decimal_Pkg for Ada application programmer convenience. There are four functions that convert between SQL_Decimal and the not null-bearing types of the SAMeDL standard numeric base domains SQL_Int and SQL_Double_Precision. Four functions are also provided to convert between SQL_Decimal and the null-bearing types of those standard base domains as well.

Two functions are provided that convert between SQL_Decimal and the not null-bearing type of the SAMeDL standard character base domain SQL_Char. Two functions are also provided to convert between SQL_Decimal and the null-bearing type of this domain.

Finally, a function that converts an SQL_Decimal object to the Ada predefined type Standard.String is also provided.

The Ada Constraint_Error exception is raised in each of the functions that convert *to* an SQL_Decimal object if the value requires more significant digits than the maximum decimal precision of the platform allows.

The Ada Constraint_Error exception is also raised in the conversion functions To_SQL_Int_Not_Null and To_SQL_Int if the decimal value is too large to be represented by an object of the SQL_Int_Not_Null and SQL_Int data types, respectively.

The SAMeDL Null_Value_Error exception is raised in each of the functions that convert to any not null-bearing type if the value being converted is the SQL *null* value.

**Attribute**

Functions identical to the attribute-like functions of the not null-bearing type are also provided for the null-bearing type. The SAMeDL Null_Value_Error exception is raised by *Width*, *Fore*, *Aft*, *Integral_Digits*, and *Scale* if the value of the SQL_Decimal object is null.

**Testing and Null_Object**

Two testing functions are implemented for objects of the null-bearing type. *Is_Null* returns the truth value *true* if the value of the object is null. Otherwise it returns *false*. *Not_Null* returns *true* if the value is not null. Otherwise it returns *false*.

The Null_SQL_Decimal operation returns an object of the null-bearing type whose value is null.

**Other Operations**

A generic subpackage called SQL_Decimal_Ops is implemented in SQL_Decimal_Pkg to provide operations that require operands of both the null-bearing and not null-bearing types of domains derived from the types of the SQL_Decimal base domain (see Section 3.3 of [11]). The SAMeDL compiler generates Ada code for the domain declaration that instantiates the generic subpackage with the not null-bearing and the null-bearing types of the domain (see Section 5.3.2.2). The subpackage exports two functions that convert between the types of the domain. The *With_Null* function converts an object of the not null-bearing type to the null-bearing type, and the *Without_Null* function converts an object of the null-bearing type to the not null-bearing type.

The generic subpackage SQL_Decimal_Ops (see Appendix C) is also instantiated with range constraints for the domain. There are many ways that range constraints can be implemented in the generic subpackage. The method chosen for SQL_Decimal domains is but one. For SQL_Decimal, the lower bound and upper bound of the range are each specified in three parts: the sign, the digits in the integral, and the scale digits of the number. It is implemented this way so that Ada will do the error checking on each particular value. The components of each number are validated against their subtype range constraints at compile time, and then the values are assembled at package elaboration time.

The generic subpackage exports a function and a procedure for each of the not null-bearing and null-bearing types of the domain that make use of the specified range constraint. The procedure is the *Assign* procedure, and it takes two arguments of the type, assigning the value of the second argument to the first argument. If the value of the second argument falls outside the specified range, the Ada Constraint_Error exception is raised. The function, *Is_In*, returns a Standard.Boolean truth value indicating whether or not the value of the input argument is within the specified range of the domain. This function is valuable in that it can be used prior to assignment of the value to an object of the domain to avoid raising an exception.

## 5.3.2. Base Domain Declaration

Once the base domain support package has been created, a SAMeDL programmer must declare the base domain in the SAMeDL. The base domain declaration supplies the SAMeDL processor with the necessary information to generate Ada code for domain declarations, and Ada and SQL code for the application/DBMS interface. The SAMeDL base domain declaration for SQL_Decimal is provided in Appendix D.[47] The following sections describe the base domain declaration for SQL_Decimal.

### 5.3.2.1. Base Domain Parameters

The SAMeDL base domain parameters specify information to be provided whenever a domain is declared from the base domain. The base domain parameters for SQL_Decimal are *scale*[48] and the range constraints for values of SQL_Decimal domains. The *scale* parameter is declared to be of data class integer, and given a default value of zero. The value supplied in the default expression must satisfy the range constraint of the *Scale_Digits* subtype in the package SQL_Decimal_Pkg because this value will be supplied as the discriminant for SQL_Decimal_Not_Null and SQL_Decimal when the types of the domain are derived from them. The range constraint parameters are decomposed into three parts, as described above in Section 5.3.1.2, so there are six parameter declarations in the base domain specification for the range constraints: *first_sign*, *first_integral*, *first_fractional*, *last_sign*, *last_integral*, and *last_fractional*. These parameters are declared to be of data class character so that expressions supplied in domain declarations for them will be compatible with the *Sign_Character* and *Numeric_String* subtypes in the package SQL_Decimal_Pkg.

If a value for scale is not specified for a domain derived from the SQL_Decimal base domain, the domain will pick up the default scale of the base domain, which is zero. If no range constraints are specified, default range constraints are supplied for the domain by the instantiation of the generic subpackage SQL_Decimal_Ops (see Appendix C).

---

[47]Refer to Section 4.1 of [18] for a complete discussion of SAMeDL base domain declarations.

[48]All base domains with data class *fixed* are required to have a scale parameter (see Section 4.1.1.1 of [18]).

### 5.3.2.2. Domain and Subdomain Patterns

Domain and subdomain patterns are specifically designed for use with a particular base domain support package. The patterns in the SQL_Decimal base domain in Appendix D are designed specifically for use with the SQL_Decimal_Pkg support package in Appendix C.

The domain template for SQL_Decimal contains three essential components. The first component consists of a type and subtype declaration. The type declaration simply derives an unconstrained discriminated record type for the domain from SQL_Decimal_Not_Null. The subtype declaration constrains this type by supplying a value for scale as the discriminant. This subtype declaration becomes the not null-bearing type of the domain. The second component also consists of a type and subtype declaration, but for the null-bearing type of the domain. The subtype declaration of this component becomes the null-bearing type of the domain. The last component of the domain template is the instantiation of the generic subpackage SQL_Decimal_Ops with the null-bearing and not null-bearing types of the domain, as well as the range constraints for values of the domain.

The derived domain template differs from the domain template only slightly. A derived domain is a domain that is derived from another domain or a subdomain, rather than from a base domain. This derivation is performed by deriving one or both of the Ada types of the domain from the Ada types of the parent domain or subdomain. For SQL_Decimal derived domains, both of the *type* declarations are derived from the unconstrained *type* declarations of the parent domain or subdomain.

The subdomain template is simpler than the other two. It consists of two subtypes and a package instantiation. The first is a subtype of the not null-bearing subtype of the parent domain or subdomain. The second is a subtype of the null-bearing subtype of the parent domain or subdomain. The package instantiation generates the appropriate conversion and assignment operators for these two new subtypes, enforcing the new range constraints.

### 5.3.2.3. Base Domain Options

As discussed above, the base domain options are necessary for the SAMeDL processor to produce Ada and SQL code for the application/DBMS interface. The options discussed here are the predefined ones (see Section 4.1.1.3 of [18]) for the SQL_Decimal base domain; however, the SAMeDL compiler implementor may require the specification of additional options to generate the necessary code for the interface (see [9]).

The first two options that are declared, **for not null type name** and **for null type name**, simply make the name of the types in the domain known to the SAMeDL. Wherever the names of the Ada types of the domain are needed in the interface code, these names will be used. The names for the types of SQL_Decimal domains are *[self]_Not_Null* and *[self]_Null*, where *[self]* is replaced with the name of the domain.

The **for data class** option specifies the data class of all objects of any domain based on the base domain. For SQL_Decimal data, the data class in the SAMeDL is *fixed*.

The **for dbms type** option specifies the SQL data type of the SQL module parameter at the concrete interface for all objects of any domain declared using this base domain. The actual Ada/SQL data type pair, or binding, is implementation-defined for each SAMeDL compiler. For the SQL_Decimal base domain example, the DBMS type was chosen to be *decimal*. The assumption is that the SAMeDL compiler will bind SQL module parameters of this type to the Max_Decimal concrete type discussed in Section 5.2.

The last four options specify the data conversions that are necessary for converting between the null-bearing, not null-bearing, and concrete data types of the domain at the application/DBMS interface. The **for conversion from dbms to not null** option specifies that the function To_SQL_Decimal_Not_Null will be used to convert the data from the concrete type (Max_Decimal, in this case) to the not null-bearing type. This function (see Appendix C) requires the specification of the scale of the object as one of the input parameters.

The **for conversion from not null to dbms** option specifies that the function To_DBMS_Type will be used to convert the data from the not null-bearing type to the concrete type, Max_Decimal.

The **for conversion from not null to null** option specifies that the function *With_Null* from the domain's instantiation of the generic subpackage SQL_Decimal_Ops will be used to convert the data from the not null-bearing type of the domain to the null-bearing type.

The **for conversion from null to not null** option specifies that the function *Without_Null* from the domain's instantiation of the generic subpackage SQL_Decimal_Ops will be used to convert the data from the null-bearing type of the domain to the not null-bearing type.

#### 5.3.2.4. Options for Other DBMS Concrete Data Types

Three base domain options in the SAMeDL base domain declaration for SQL_Decimal must be examined when developing support for a concrete representation other than the packed decimal representation discussed in this report. These options are the **for dbms type**, **for conversion from dbms to not null**, and **for conversion from not null to dbms** options, and they may need to be changed to indicate the use of a different concrete type.

If the concrete type to be supported is the SQL_Standard.Int data type, for example, the **for dbms type** option must specify *integer* instead of *decimal*. Additionally, functions must be implemented to convert between SQL_Standard.Int and SQL_Decimal_Not_Null. The **for conversion from dbms to not null** option must specify the function that converts from the concrete type to the not null-bearing type, and the **for conversion from not null to dbms** must specify the function that converts from the not null-bearing type to the concrete type. If these two functions were named *To_SQL_Decimal_Not_Null* and *To_SQL_Standard_Int*, respectively, then the specification of the three new options would appear as follows.

```
for dbms type use integer;
for conversion from dbms to not null use function
    'To_SQL_Decimal_Not_Null';
for conversion from not null to dbms use function
    'To_SQL_Standard_Int';
```

## 5.4. SQL Decimal Domain Declarations

After the base domain has been declared, it can be used to declare domains in the SAMeDL. Typical decimal domain declarations based on the SQL_Decimal base domain discussed in this report are provided in the following example. Assume that the domain declarations are declared within a definitional module not shown here, and that the definitional module in which the SQL_Decimal base domain is declared is directly visible at the point the domains are declared.

```
domain Feet is new SQL_Decimal (
    scale            => 2,
    first_sign       => '+',
    first_integral   => '0',
    first_fractional => '00',
    last_sign        => '+',
    last_integral    => '100',
    last_fractional  => '00');

domain Square_Feet is new SQL_Decimal (
    scale            => 2,
    first_sign       => '+',
    first_integral   => '0',
    first_fractional => '00',
    last_sign        => '+',
    last_integral    => '10000',
    last_fractional  => '00');
```

The SAMeDL processor will then produce the following Ada code for the domain declarations, based on the domain template specified in the SQL_Decimal base domain declaration.

```
type FeetNN_Base is new SQL_Decimal_Not_Null;
subtype Feet_Not_Null is FeetNN_Base (2);
type Feet_Base is new SQL_Decimal;
subtype Feet_Type is Feet_Base (2);
package Feet_Ops is new SQL_Decimal_Ops(
    Feet_Base, FeetNN_Base,
    in_scale => 2
    , first_sign => '+',
    first_integral => '0',
    first_fractional => '00'
    , last_sign => '+',
    last_integral => '100',
    last_fractional => '00');

type Square_FeetNN_Base is new SQL_Decimal_Not_Null;
subtype Square_Feet_Not_Null is Square_FeetNN_Base (2);
type Square_Feet_Base is new SQL_Decimal;
subtype Square_Feet_Type is Square_Feet_Base (2);
package Square_Feet_Ops is new SQL_Decimal_Ops(
    Square_Feet_Base, Square_FeetNN_Base,
    in_scale => 2
    , first_sign => '+',
```

```
first_integral => ’0’,
first_fractional => ’00’
, last_sign => ’+’,
last_integral => ’10000’,
last_fractional => ’00’);
```

# References


**1.** *Ada 9X Project Report - Requirements Workshop.* Office of the Undersecretary of Defense for Acquisition, June 1989.

**2.** *Ada 9X Project Report DRAFT Mapping Document.* Ada 9X Project Office, February 1991.

**3.** *Reference Manual for the Ada Programming Language.* Ada Joint Program Office, January 1983.

**4.** *Database Language - SQL with Integrity Enhancement.* American National Standards Institute, 1989. X3.135-1989.

**5.** *Database Language - Embedded SQL X3.168-1989.* American National Standards Institute, 1989.

**6.** Booch Grady. *Software Engineering with Ada.* The Benjamin/Cummings Publishing Company Inc., 1983.

**7.** Brykczynski, B. R.; Friedman, F. Preliminary Version: Ada/SQL: A Standard, Portable, Ada-DBMS Interface. Tech. Rept. P-1944, Institute for Defense Analyses, Alexandria, Virginia, July 1986.

**8.** Chastek, Gary, Graham, Marc H., and Zelesnik Gregory. The SQL Ada Module Description Language - SAMeDL. Tech. Rept. CMU/SEI-90-TR-26, Software Engineering Institute, November 1990.

**9.** Chastek, Gary, and Graham, Marc H. Rationale for SQL Ada Module Description Language - SAMeDL. Tech. Rept. CMU/SEI-91-TR-4, Software Engineering Institute, 1991.

**10.** *Database Language SQL.* International Standards Organization, April 1991. DIS 9075:199x(E).

**11.** Graham, Marc H. Guidelines for the Use of the SAME. Tech. Rept. CMU/SEI-89-TR-16, ADA228027, Software Engineering Institute, May 1989.

**12.** *IBM DATABASE2 Reference.* International Business Machines (IBM) Corporation, March 1986.

**13.** *IBM DATABASE2 Data Base Planning and Administration Guide.* International Business Machines (IBM) Corporation, May 1987.

**14.** *INGRES/EMBEDDED SQL User's Guide and Reference Manual.* Relational Technology Inc., August 1989.

**15.** *INGRES/SQL Reference Manual.* Relational Technology Inc., August 1989.

**16.** Melton J. (Editor). *Database Language SQL2 (ISO working draft).* International Organization for Standardization and American National Standards Institute X3H2, 1990.

**17.** *Pro*C User's Guide (Version 1.1).* Oracle Corporation, April 1987.

**18.** *SQL Ada Module Description Language.* ISO/JTC1/SC22/WG9, May 1991.

**19.** *SQL*Plus Reference Guide (Version 2.0).* Oracle Corporation, July 1987.

**20.** Ullman, Jeffrey D.. *Principles of DATABASE SYSTEMS.* Computer Science Press, Inc., 1982.

# Appendix A: Procedural Interface SAMeDL and Ada Code

```
with Dynamic_Domains; use Dynamic_Domains;
extended abstract module Dynamic_Stmts is

    extended procedure Allocate_Descriptor
        (Descriptor       : Descriptor_Name;
         Max_Variables    : Variable_Occurrences)
    is
        ALLOCATE DESCRIPTOR Descriptor WITH MAX Max_Variables;

    extended procedure Deallocate_Descriptor
        (Descriptor       : Descriptor_Name)
    is
        DEALLOCATE DESCRIPTOR Descriptor;

    extended procedure Get_Count
        (Descriptor       : Descriptor_Name;
         Nbr_Variables    : out Variable_Occurrences)
    is
        GET DESCRIPTOR Descriptor Nbr_Variables = COUNT;

    extended procedure Get_Parameter
        (Descriptor       : Descriptor_Name;
         Variable         : Variable_Occurrences;
         Parameter        : out Dynamic_Parameter)
    is
        GET DESCRIPTOR Descriptor VALUE Variable;

    extended procedure Set_Count
        (Descriptor       : Descriptor_Name;
         Nbr_Variables    : Variable_Occurrences)
    is
        SET DESCRIPTOR Descriptor COUNT = Nbr_Variables;

    extended procedure Set_Parameter
        (Descriptor       : Descriptor_Name;
         Variable         : Variable_Occurrences;
         Parameter        : Dynamic_Parameter)
    is
        SET DESCRIPTOR Descriptor VALUE Variable;

    extended procedure Prepare
        (Identifier   : Dynamic_Statement_Identifier;
         Statement    : Dynamic_Statement)
    is
        PREPARE Identifier FROM Statement;

    extended procedure Deallocate_Prepare
        (Identifier   : Dynamic_Statement_Identifier)
    is
        DEALLOCATE PREPARE Identifier;
```

```
extended procedure Describe_Input
    (Identifier   : Dynamic_Statement_Identifier;
     Descriptor   : Descriptor_Name)
is
    DESCRIBE INPUT Identifier USING Descriptor;

extended procedure Describe_Output

    (Identifier   : Dynamic_Statement_Identifier;
     Descriptor   : Descriptor_Name)
is
    DESCRIBE OUTPUT Identifier USING Descriptor;

extended procedure Execute
    (Identifier     : Dynamic_Statement_Identifier)
is
    EXECUTE Identifier;

extended procedure Execute_with_Descriptor
    (Identifier   : Dynamic_Statement_Identifier;
     Descriptor   : Descriptor_Name)
is
    EXECUTE Identifier USING Descriptor;

extended procedure Execute_Immediate
    (Identifier     : Dynamic_Statement_Identifier)
is
    EXECUTE IMMEDIATE Identifier;

extended procedure Allocate_Cursor
    (Identifier   : Dynamic_Statement_Identifier;
     Cursor_Identifier : Dynamic_Cursor_Identifier)
is
    ALLOCATE Cursor_Identitifer CURSOR
        FOR Identifier;

extended procedure Allocate_Cursor_Scroll
    (Identifier   : Dynamic_Statement_Identifier;
     Cursor_Identifier : Dynamic_Cursor_Identifier)
is
    ALLOCATE Cursor_Identitifer SCROLL CURSOR
        FOR Identifier;

extended procedure Open
    (Cursor_Identifier : Dynamic_Cursor_Identifier)
is
    OPEN Cursor_Identifier;

extended procedure Open_with_Descriptor
    (Cursor_Identifier : Dynamic_Cursor_Identifier;
     Descriptor        : Descriptor_Name)
is
    OPEN Cursor_Identifier USING Descriptor;

extended procedure Fetch
    (Cursor_Identifier : Dynamic_Cursor_Identifier;
```

```
         Descriptor        : Descriptor_Name)
    is
        FETCH Cursor_Identifier USING Descriptor
        status Standard_Map;

    extended procedure Fetch_First
        (Cursor_Identifier : Dynamic_Cursor_Identifier;
         Descriptor        : Descriptor_Name)
    is
        FETCH FIRST FROM Cursor_Identifier USING Descriptor
        status Standard_Map;

    extended procedure Fetch_Prior
        (Cursor_Identifier : Dynamic_Cursor_Identifier;
         Descriptor        : Descriptor_Name)
    is
        FETCH PRIOR FROM Cursor_Identifier USING Descriptor
        status Standard_Map;

-- other fetch procedures for other <fetch orientation>s

    extended procedure Close
        (Cursor_Identifier : Dynamic_Cursor_Identifier)
    is
        CLOSE Cursor_Identifier;


end Dynamic_Stmts;
```

**Figure A-1:** Procedure Interface for Dynamic SQL

```
with Dynamic_Base_Domains, SAMeDL_Standard;
use  Dynamic_Base_Domains, SAMeDL_Standard;
definition module Dynamic_Domains is
    domain Dynamic_Char is new SQL_Unconstrained_Char;
    domain Dynamic_Int is new SQL_Int;
    domain Dynamic_Smallint is new SQL_Smallint;
    domain Dynamic_Real is new SQL_Real;
    domain Dynamic_Double_Precision is new SQL_Double_Precision;

    domain Descriptor_Name is new SQL_Char
            Length => implementation defined
    domain Dynamic_Statement_Identifier is new SQL_Char
            Length => implementation defined
    domain Dynamic_Statement is new SQL_Char
            Length => implementation defined
    domain Dynamic_Cursor_Identifier is new SQL_Char
            Length => implementation defined

    domain Variable_Occurrences is new SQL_Int not null
            (First => 1, Last => implementation defined);

    domain Dynamic_Parameter is new Dynamic_Parameter_Base;
end Dynamic_Domains;
```

**Figure A-2:** Domains for Procedures in Module Dynamic_Stmts

```
with SAMeDL_Standard; use SAMeDL_Standard;
definition module Dynamic_Base_Domains is

    base domain SQL_Unconstrained_Char is

        domain pattern is
            'type [self]_Not_Null is new SQL_Char_Not_Null;'
            'type [self]_Type is new SQL_Char'
            'package [self]_Ops is new SQL_Char_Ops('
            '     [self]_Type, [self]_Not_Null);'
        end pattern;
            -- and so forth for the other patterns

        for not null type name use '[self]_Not_Null';
        for null type name use '[self]_Type';
        for data class use character;
        for dbms type use varchar;
        for conversion from dbms to not null use type mark;
        for conversion from not null to null use function
            '[self]_Ops.With_Null';
        for conversion from null to not null use function
            '[self]_Ops.Without_Null';
        for conversion from not null to dbms use type mark;
    end SQL_Unconstrained_Char;

    base domain Dynamic_Parameter_Base is
        domain pattern is
'type SQL_Dynamic_Datatypes is'
'    (Not_Specified,'
'     Dynamic_Char, Dynamic_Int, Dynamic_Smallint, '
'     Dynamic_Real, Dynamic_Double_Precision);'
''
'-- access types access null bearing types in Dynamic_DataTypes_Pkg'
'type Char_Access is access Dynamic_Char_Type;'
'type Int_Access is access Dynamic_Int_Type;'
'type Smallint_Access is access Dynamic_Smallint_Type;'
'type Real_Access is access Dynamic_Real_Type;'
'type Double_Precision_Access is access Dynamic_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_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;'
''
        end pattern;
            -- the options are also unimportant : SEE TEXT
end Dynamic_Base_Domains;
```

**Figure A-3:** Base Domains SQL_Unconstrained_Char and Dynamic_Parameter_Base

```
with SQL_Char_Pkg, SQL_Int_Pkg, SQL_Smallint_Pkg,
        SQL_Real_Pkg, SQL_Double_Precision_Pkg;
use  SQL_Char_Pkg, SQL_Int_Pkg, SQL_Smallint_Pkg,
        SQL_Real_Pkg, SQL_Double_Precision_Pkg;
package Dynamic_Domains is

    type Dynamic_Char_Not_Null is new SQL_Char_Not_Null;
    type Dynamic_Char_Type is new SQL_Char;
    package Dynamic_Char_Ops is new SQL_Char_Ops(
        Dynamic_Char_Type, Dynamic_Char_Not_Null);

    type Dynamic_Int_Not_Null is new SQL_Int_Not_Null;
    type Dynamic_Int_Type is new SQL_Int;
    package Dynamic_Int_Ops is new SQL_Int_Ops(
        Dynamic_Int_Type, Dynamic_Int_Not_Null);

    type Dynamic_Smallint_Not_Null is new SQL_Smallint_Not_Null;
    type Dynamic_Smallint_Type is new SQL_Smallint;
    package Dynamic_Smallint_Ops is new SQL_Smallint_Ops(
        Dynamic_Smallint_Type, Dynamic_Smallint_Not_Null);

    type Dynamic_Real_Not_Null is new SQL_Real_Not_Null;
    type Dynamic_Real_Type is new SQL_Real;
    package Dynamic_Real_Ops is new SQL_Real_Ops(
        Dynamic_Real_Type, Dynamic_Real_Not_Null);

    type Dynamic_Double_Precision_Not_Null
        is new SQL_Double_Precision_Not_Null;
    type Dynamic_Double_Precision_Type
        is new SQL_Double_Precision;
    package Dynamic_Double_Precision_Ops
        is new SQL_Double_Precision_Ops(
            Dynamic_Double_Precision_Type,
            Dynamic_Double_Precision_Not_Null);

    type Descriptor_NameNN_Base is new SQL_Char_Not_Null;
    subtype Descriptor_Name_Not_Null
        is Descriptor_NameNN_Base (1 .. implementation defined);
    type Descriptor_Name_Base is new SQL_Char;
    subtype Descriptor_Name_Type
        is Descriptor_Name_Base (Descriptor_Name_Not_Null'Length);
    package Descriptor_Name_Ops is new SQL_Char_Ops(
        Descriptor_Name_Base, Descriptor_NameNN_Base);

    type Dynamic_Statement_IdentifierNN_Base is new SQL_Char_Not_Null;
    subtype Dynamic_Statement_Identifier_Not_Null
        is Dynamic_Statement_IdentifierNN_Base
            (1 .. implementation defined);
    type Dynamic_Statement_Identifier_Base is new SQL_Char;
    subtype Dynamic_Statement_Identifier_Type
        is Dynamic_Statement_Identifier_Base
            (Dynamic_Statement_Identifier_Not_Null'Length);
    package Dynamic_Statement_Identifier_Ops is new SQL_Char_Ops(
        Dynamic_Statement_Identifier_Base,
        Dynamic_Statement_IdentifierNN_Base);
```

```
type Dynamic_StatementNN_Base is new SQL_Char_Not_Null;
subtype Dynamic_Statement_Not_Null
    is Dynamic_StatementNN_Base (1 .. implementation defined);
type Dynamic_Statement_Base is new SQL_Char;
subtype Dynamic_Statement_Type
    is Dynamic_Statement_Base (Dynamic_Statement_Not_Null'Length);
package Dynamic_Statement_Ops is new SQL_Char_Ops(
    Dynamic_Statement_Base,
    Dynamic_StatementNN_Base);

type Dynamic_Cursor_IdentifierNN_Base is new SQL_Char_Not_Null;
subtype Dynamic_Cursor_Identifier_Not_Null
    is Dynamic_Cursor_IdentifierNN_Base
        (1 .. implementation defined);
type Dynamic_Cursor_Identifier_Base is new SQL_Char;
subtype Dynamic_Cursor_Identifier_Type
    is Dynamic_Cursor_Identifier_Base
        (Dynamic_Cursor_Identifier_Not_Null'Length);
package Dynamic_Cursor_Identifier_Ops is new SQL_Char_Ops(
    Dynamic_Cursor_Identifier_Base,
    Dynamic_Cursor_IdentifierNN_Base);

type Variable_Occurrences_Not_Null is new SQL_Int_Not_Null
    range 1 .. implementation defined;
type Variable_Occurrences_Type is new SQL_Int;
package Variable_Occurrence_Ops is new SQL_Int_Ops(
    Variable_Occurrences_Type, Variable_Occurrences_Not_Null);

type SQL_Dynamic_Datatypes is
(Not_Specified,
 Dynamic_Char, Dynamic_Int, Dynamic_Smallint,
 Dynamic_Real, Dynamic_Double_Precision);

type Char_Access is access Dynamic_Char_Type;
type Int_Access is access Dynamic_Int_Type;
type Smallint_Access is access Dynamic_Smallint_Type;
type Real_Access is access Dynamic_Real_Type;
type Double_Precision_Access is access Dynamic_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_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;

end Dynamic_Domains;
```

**Figure A-4:** The Domain Declarations in Ada

```
with Dynamic_Domains; use Dynamic_Domains;
package Dynamic_Stmts is

    procedure Allocate_Descriptor
        (Descriptor      : Descriptor_Name_Type;
         Max_Variables   : Variable_Occurrences_Not_Null);

    procedure Deallocate_Descriptor
        (Descriptor      : Descriptor_Name_Type);

    procedure Get_Count
        (Descriptor      : Descriptor_Name_Type;
         Nbr_Variables   : out Variable_Occurrences_Not_Null);

    procedure Get_Parameter
        (Descriptor      : Descriptor_Name_Type;
         Variable        : Variable_Occurrences_Not_Null;
         Parameter       : out Dynamic_Parameter);

    procedure Set_Count
        (Descriptor      : Descriptor_Name_Type;
         Nbr_Variables   : Variable_Occurrences_Not_Null);

    procedure Set_Parameter
        (Descriptor      : Descriptor_Name_Type;
         Variable        : Variable_Occurrences_Not_Null;
         Parameter       : Dynamic_Parameter);

    procedure Prepare
        (Identifier  : Dynamic_Statement_Identifier_Type;
         Statement   : Dynamic_Statement_Type);

    procedure Deallocate_Prepare
        (Identifier  : Dynamic_Statement_Identifier_Type);

    procedure Describe_Input
        (Identifier   : Dynamic_Statement_Identifier_Type;
         Descriptor   : Descriptor_Name_Type);

    procedure Describe_Output

        (Identifier   : Dynamic_Statement_Identifier_Type;
         Descriptor   : Descriptor_Name_Type);

    procedure Execute
        (Identifier     : Dynamic_Statement_Identifier_Type);

    procedure Execute_with_Descriptor
        (Identifier   : Dynamic_Statement_Identifier_Type;
         Descriptor   : Descriptor_Name_Type);

    procedure Execute_Immediate
        (Identifier     : Dynamic_Statement_Identifier_Type);

    procedure Allocate_Cursor
        (Identifier   : Dynamic_Statement_Identifier_Type;
```

```
         Cursor_Identifier : Dynamic_Cursor_Identifier_Type);

    procedure Allocate_Cursor_Scroll
        (Identifier   : Dynamic_Statement_Identifier_Type;
         Cursor_Identifier : Dynamic_Cursor_Identifier_Type);

    procedure Open
        (Cursor_Identifier : Dynamic_Cursor_Identifier_Type);

    procedure Open_With_Descriptor
        (Cursor_Identifier : Dynamic_Cursor_Identifier_Type;
         Descriptor        : Descriptor_Name_Type);

    procedure Fetch
        (Cursor_Identifier : Dynamic_Cursor_Identifier_Type;
         Descriptor        : Descriptor_Name_Type);

    procedure Fetch_First
        (Cursor_Identifier : Dynamic_Cursor_Identifier_Type;
         Descriptor        : Descriptor_Name_Type);

    procedure Fetch_Prior
        (Cursor_Identifier : Dynamic_Cursor_Identifier_Type;
         Descriptor        : Descriptor_Name_Type);

-- other fetch procedures for other <fetch orientation>s

    procedure Close
        (Cursor_Identifier : Dynamic_Cursor_Identifier_Type);

end Dynamic_Stmts;
```

**Figure A-5:** The Procedural Interface in Ada

```
with Concrete_Dynamic_Stmts, SQL_Communications_Pkg, SQL_Standard,
    SQL_Database_Error_Pkg;
use SQL_Communications_Pkg ,SQL_Database_Error_Pkg, SQL_Standard;
package body Dynamic_Stmts is

    package CDS renames Concrete_Dynamic_Stmts;

    Character :   constant := 1; -- these are the codes
    integer :     constant := 4; -- specified by SQL2
    Smallint :    constant := 5;
    Real :        constant := 7;
    DoublePrecision :
                  constant := 8;

    subtype Coded_Data_Types is Standard.integer
        range Character .. DoublePrecision;

    procedure Get_Parameter
        (Descriptor      : Descriptor_Name_Type;
         Variable        : Variable_Occurrences_Not_Null;
         Parameter       : out Dynamic_Parameter) is

    Type_Variable : Coded_Data_Types;
    Length_Variable : Standard.integer;

    begin
        CDS.Get_Type(Descriptor => Descriptor,
                     Variable => Variable,
                     Parm_Type => Type_Variable,
                     SQLCODE => SQLCODE);
        if SQLCODE /= 0 then
            Process_Database_Error;
            raise SQL_Database_Error;
        end if;
        case Type_Variable is
            when Character =>
                CDS.Get_Length(Descriptor => Descriptor,
                               Variable => Variable,
                               Parm_Length => Length_Variable,
                               SQLCODE => SQLCODE);
                if SQLCODE /= 0 then
                    Process_Database_Error;
                    raise SQL_Database_Error;
                end if;
                declare
                    Character_Buffer :
                       SQL_Standard.Char (1 .. Length_Variable);
                    Indicator_Buffer : SQL_Standard.Indicator_Type;

                    Result : Dynamic_Parameter :=
                            (SQLTYPE => Dynamic_Char,
                             Char_Value => new Dynamic_Char_Type(
                                                 Length => Length_Variable)
                            );
                begin
                    CDS.Get_Value(Descriptor => Descriptor,
```

```
                                  Variable => Variable,
                                  Value => Character_Buffer,
                                  Indicator => Indicator_Buffer,
                                  SQLCODE => SQLCODE);
                    if SQLCODE /= 0 then
                        Process_Database_Error;
                        raise SQL_Database_Error;
                    end if;
                    if Indicator_Buffer >= 0 then
                        assign(Result.Char_Value.all,
                               Dynamic_Char_Ops.With_Null(
                                  Dynamic_Char_Not_Null(
                                    Character_Buffer
                               )));
                    else
                        assign(Result.Char_Value.all, Null_SQL_Char);
                    end if;
                    Parameter := Result;
                end;
            -- other cases of Coded_Data_Types
        end case;
    end Get_Parameter;
end Dynamic_Stmts;
```

**Figure A-6:** A Portion of the Body of the Procedural Interface

# Appendix B: Concrete_Decimal

The following is the Ada package specification for Concrete_Decimal, the package that contains the Ada concrete type definition for *Max_Decimal* used in the decimal binding discussed throughout this paper.

```
package Concrete_Decimal 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;

    -- Max_Decimal is the Ada concrete type definition used in
    --    a decimal binding where the decimal parameter has a
    --    precision of MAX_DIGITS for representation of the BCD
    --    value
    type Max_Decimal is private;

private

    -- 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 representation clause below guarantees an array
    --    component of eight bits
    for Digit'size use 8;

    -- MAX_SIZE is the number of array positions needed for
    --    the Max_Decimal type
    -- since each BCD digit can be represented by 4 bits,
    --    the total number of bits for the numeric part can
    --    be calculated by MAX_DIGITS * 4;
    -- this result is divided by the number of bits required
    --    for a single array position, which yields the number
    --    of array positions needed for the BCD number
    -- the result is incremented by one to accomodate the sign
    MAX_SIZE : constant Integer := ((4 * (MAX_DIGITS)) / Digit'size) + 1;

    -- Max_Decimal is the Ada concrete type definition used in
    --    a decimal binding where the decimal parameter has a
    --    precision of MAX_DIGITS for representation of the BCD
    --    value
    type Max_Decimal is array (1..MAX_SIZE) of Digit;

end Concrete_Decimal;
```

# Appendix C: SQL_Decimal_Pkg

The following Ada package specification is the package specification for the SQL_Decimal base domain support package that has been discussed throughout this paper. The package body for SQL_Decimal_Pkg can be found in Section 19 of Appendix C of [11]. Additionally, the assembler subroutines necessary to implement the packed decimal arithmetic instructions on two platforms are supplied in Sections 20 and 21 of that same appendix.

```
with SQL_Standard;
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;
with Concrete_Decimal; use Concrete_Decimal;
package SQL_Decimal_Pkg is

    use SQL_Standard.Character_Set;

    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 SQL_Standard.Character_Type
        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
--    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 between
--    SQL_Decimal_Not_Null and the concrete type.
function To_SQL_Decimal_Not_Null (Scale : Decimal_Digits;
                                  Value : Max_Decimal)
    return SQL_Decimal_Not_Null;
function To_DBMS_Type (Right : SQL_Decimal_Not_Null)
    return Max_Decimal;
pragma INLINE(To_DBMS_Type);

-- 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 : 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_Type);

    end SQL_Decimal_Ops;

private

    -- the not null-bearing type
    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;

    -- the null-bearing type
    type SQL_Decimal(Scale : Decimal_Digits) is record
        Is_Null : Boolean := True;
        Value   : SQL_Decimal_Not_Null(Scale);
    end record;

end SQL_Decimal_Pkg;
```

# Appendix D: SQL_Decimal

The following SAMeDL base domain declaration for SQL_Decimal is the declaration that is discussed in Section 5.3.2 of this paper.

```
base domain SQL_Decimal
    (scale            : integer := 0;
     first_sign       : character;
     first_integral   : character;
     first_fractional : character;
     last_sign        : character;
     last_integral    : character;
     last_fractional  : character)
is

    domain pattern is
        'type [self]NN_Base is new SQL_Decimal_Not_Null;'
        'subtype [self]_Not_Null is [self]NN_Base ([scale]);'
        'type [self]_Base is new SQL_Decimal;'
        'subtype [self]_Type is [self]_Base ([scale]);'
        'package [self]_Ops is new SQL_Decimal_Ops ('
        '[self]_Base, [self]NN_Base,'
        'in_scale => [scale]'
        '{, first_sign => [first_sign], '
        'first_integral => [first_integral], '
        'first_fractional => [first_fractional]}'
        '{, last_sign => [last_sign], '
        'last_integral => [last_integral], '
        'last_fractional => [last_fractional]});'
    end pattern;

    derived domain pattern is
        'type [self]NN_Base is new [parent]NN_Base;'
        'subtype [self]_Not_Null is [self]NN_Base ([scale]);'
        'type [self]_Base is new [parent]_Base;'
        'subtype [self]_Type is [self]_Base ([scale]);'
        'package [self]_Ops is new SQL_Decimal_Ops ('
        '[self]_Base, [self]NN_Base,'
        'in_scale => [scale]'
        '{, first_sign => [first_sign], '
        'first_integral => [first_integral], '
        'first_fractional => [first_fractional]}'
        '{, last_sign => [last_sign], '
        'last_integral => [last_integral], '
        'last_fractional => [last_fractional]});'
    end pattern;

    subdomain pattern is
        'subtype [self]_Not_Null is [parent]_Not_Null;'
        'subtype [self]_Type is [parent]_Type;'
        'package [self]_Ops is new SQL_Decimal_Ops ('
        '[self]_Type, [self]_Not_Null,'
        'in_scale => [scale]'
        '{, first_sign => [first_sign], '
        'first_integral => [first_integral], '
```

```
        'first_fractional => [first_fractional]}'
        '{, last_sign => [last_sign], '
        'last_integral => [last_integral], '
        'last_fractional => [last_fractional]});'
    end pattern;

    for not null type name use '[self]_Not_Null';
    for null type name use '[self]_Type';
    for data class use fixed;
    for dbms type use decimal;
    for conversion from dbms to not null use function
        'To_SQL_Decimal_Not_Null';
    for conversion from not null to null use function
        '[self]_Ops.With_Null';
    for conversion from null to not null use function
        '[self]_Ops.Without_Null';
    for conversion from not null to dbms use function
        'To_DBMS_Type';

end SQL_Decimal;
```

# Table of Contents

# List of Figures