Technical Report

CMU/SEI-92-TR-16
ESC-92-TR-16

Rationale for SQL Ada Module Description Language SAMeDL

Version 2.0

Gary J. Chastek
Marc H. Graham
Gregory Zelesnik

October 1992

Technical Report
CMU/SEI-92-TR-16
ESC-92-TR-16
October 1992

Rationale for SQL Ada Module Description Language SAMeDL

Version 2.0

Gary J. Chastek
Marc H. Graham
Gregory Zelesnik

Binding of Ada and SQL Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Rationale for SQL Ada Module Description Language SAMeDL

Abstract: The SQL Ada Module Description Language, SAMeDL, is a language for the specification of Abstract Interfaces as delineated by the SQL Ada Module Extensions (SAME) methodology. The language is formally defined in the SAMeDL Reference Manual [Chastek 90]. This document is a companion to the Reference Manual. Whereas the Reference Manual is meant to be precise, the Rationale is meant to be clear.

An explanation of the problem solved by the SAMeDL is given. The creation of a new language is justified and the underlying principles of that language are described. Crucial issues in the language are then explained. These include:

- The form and meaning of identifiers in the SAMeDL.
- The role of and procedures for data definition in the SAMeDL. This includes support for enumerations and constants in the SAMeDL.
- The typing rules of the SAMeDL.

The proposed use of some SAMeDL features is also illustrated. These include Standard Post Processing and User Defined Base Domains.

This document is a revision of an earlier technical report, SEI/CMU-91-TR-4 [Chastek2 91]. The revision consists of the addition of a reference model of database programming language interfaces to Section 1.3. The reference model gives a context for the survey of Ada SQL interface solutions given in Section 1.3.

1. Overview

The lack of an acceptable standard interface between the programming language Ada and the database language SQL is "the greatest impediment to the further spread of Ada" [Ichbiah 90]. The SQL Ada Module Description Language, the SAMeDL, has been designed to fill this gap. This document describes the SAMeDL and explains the ideas on which it is based.

A complete description of the SAMeDL can be found in [Chastek 90]. The SAMeDL is designed to facilitate the construction of Ada database applications that conform to the SQL Ada Module Extensions (SAME) architecture as described in [Graham 89]. Together these documents present a complete explanation of the Ada SQL binding [Chastek 90] and [Graham 89]. These documents presuppose the Ada and SQL standards [ada 83] [ansiinteg 89].

1.1. The Problem

The problem is the specification of an interface, preferably a standard, which allows application programs written in Ada to access and manipulate data stored in a database under the control of a commercial off the shelf (COTS) database management system (DBMS) that responds to directives written in the database language SQL. The problem does *not* require the solution to be a persistent storage mechanism, which is in some sense ideal for Ada programs.

1.2. Properties of An Acceptable Solution

The problem as stated allows for a myriad of solutions and a myriad of solutions has indeed been proposed. This section describes the issues by which any solution may be judged. It is based on the discussion in [Engle 87].

There are requirements that any solution must meet in order to be considered acceptable. This list presents the requirements.

- **Portability:** A primary purpose of standards promulgation is the provision of portability for user written software. The Ada SQL interface should support the portability of all application programmer written code among target hardware, operating systems, Ada compilers and SQL engines. Portability is made more difficult by the minimalist nature of the current SQL standard [ansiinteg 89]. Most if not probably all DBMS offer functionality not covered by the standard.
- **Shared data - interoperability:** There are multi language shops which have a large investment in non Ada SQL applications which they will not be willing to discard. The database must be accessible from and updatable by non Ada programs. These programs will include the utilities provided by the DBMS vendor, such as forms interfaces and data entry utilities.
- **Efficiency:** The solution should not impose excessive compile or run time costs. The standard must avoid mandating any feature for which no efficient implementation exists.
- **Range of acceptance:** The solution must be acceptable to a large segment of the user community. Should the Department of Defense mandate an interface which is rejected by the wider community, it will lose much of the benefits of standardization. Vendors will be reluctant to produce such an interface, thereby diminishing competition. Programmers will be less likely to be trained in the interface, thereby increasing training costs.

This list contains some of the issues which all Ada SQL interface proposals face.

- **Basic, atomic, or scalar types:** These are the machine representation oriented types of the two languages. The task of reconciling the representations of scalar types shared by the two systems (e.g., how many bits are there in a database integer?) has been accomplished by the ANSI SQL standard [Dewar 89]. But what of the types not shared such as DECIMAL in SQL, or enumeration in Ada? Should the interface support both, either, or neither?

What about operations on the shared types? For example, Ada and SQL use different rules for string comparisons.

- **Missing information:** How is the SQL notion of null value to be handled in the Ada binding and in the applications it supports?
- **Strong, application types:** Should an interface provide a strong typing discipline similar to Ada's? Can the typing scheme allow for application oriented typing, types such as EMPLOYEE_NUMBER and PART_NUMBER instead of INTEGER?
- **Exceptional conditions:** How are database error conditions to be signaled to the application? Should all exceptional conditions be signaled as Ada exceptions? Even for expected conditions, such as "no record found?" How many different Ada exceptions should there be and how should the mapping from database condition to Ada exception be specified?

1.3. A Reference Model and Survey of Ada SQL Interface Solutions

This section contain a model of the solution space for Ada SQL Interfaces and a survey of some of the solutions that are in use or have been proposed. An outline of the model is given by the layered diagram in Figure 1-1.[1]

At the topmost level, the Application Specific Interface provides services specific to the application using the interface. Operations at this level are in application terms; collecting and storing information about objects, events and concepts of interest to the application (e.g., hiring a new employee, producting invoices for certain customers, etc.). All applications contain such an interface, although not all applications identify it as such.

At the lowest level, the Call Level Interfact, sometimes called an Application Programming Interface, or API, consists of a fixed set of procedure declarations in a given language. These procedures operate at a low level of abstraction in the sense that they maniuplate small details of the aplication/database interaction that are not visible to programmers working at higher levels. The API is usually specific to a particular Database Management System and is the target for processors at the higher levels.

Two intermediate levels are identified: the Abstract and the Concrete. Each level is further divided into three sub-categories: Modular, Embedded, and Other. Abstract and Concrete interfaces are distinguished by the typing paradigm each presents to the next higher layer. Abstract interfaces manipulate objects with abstract, application-oriented types; concrete interfaces deal with objects having concrete, machine-oriented types.

---

[1]This model has been created in collaboration with the SQL Rapporteur Group of ISO SC22/WG9, the international standards organization for Ada. The authors would like to particularly thank Rudolf Landwehr, Steve Michell, Jim Moore, and Jean-Pierre Rosen.

Figure 1-1: Ada SQL Interface Reference Model

Modular interfaces textually separate the database language statements from the programming language statements. Embedded interfaces textually intermix database language and programming language statements. As the name implies, the Other category hold interface styles that do not fit the main categories. The following list gives examples in four of the six categories of the intermediate layers. We know of no examples in the remaining categories.

- **Abstract Modular**. The SAMeDL is an abstract, modular interface.
- **Abstract Other**. Ada/SQL [Brykczynski1 86]; SQL_ARMAda [Baer 90]. These are discussed in Section 1.3.2.
- **Concrete Modular**. The SQL module language [Melton2 92].
- **Concrete Embedded**. Embedded SQL, discussed in Section 1.3.1 below.

The following sections describe these examples. The discussion begins with the traditional solution, embedded SQL. It then describes the solutions, called "Abstract Other" above, that have been proposed specifically for Ada. Finally, modular solutions, both abstract and concrete, are described. The proposals are then rated against the criteria and issues in Section 1.2.

1.3.1. Embedded SQL

Traditionally, application programming languages were given access to a DBMS via what has been called a "data sublanguage." (See Chapter 3 of [Date 75].) The earliest example of such a sublanguage to be standardized was the extension to COBOL often called the CODASYL DBMS [Cobol 78]. A sublanguage is a collection of statement kinds, often called verbs or operations but given in a syntax suitable for describing programming constructs (e.g., BNF). The addition of such syntax to COBOL, for example, results effectively in a new language. In the case of COBOL and the CODASYL DBMS, this new language is reasonably coherent; the underlying models, procedurality, record-at-a-time processing, etc., are compatible. In the case of Ada and the relational language SQL, the underlying models are quite distinct. Ada is a third generation programming language in which algorithms are described. SQL is a data manipulation language in which the intended results are described, the algorithm which produces the result being left to the DBMS.

To avoid the expense of creating an entire compiler, the early implementations of data sublanguages produced so-called pre-compilers [idms 78] [quel 76] [sysr 76]. These pre-compilers remove operations in the data sublanguage from the program text and replace them with text legal in the base language, generally with procedure calls. The process is diagrammed in Figure 1-2. This pre-compilation or "embedding" of the data sublanguage has long been the *de facto* standard for the binding of programming languages and DBMS. The SQL standard contains embedded interfaces for six programming languages, including Ada, which are in concept identical to the interfaces first proposed more than twenty years ago [ANSI2 86].

Much has been written about the disadvantages of embedding SQL into Ada [Donaho 87] [Boyd 87] [Brykczynski2 87] [Engle 87] [Vasilescu 90]. These authors have recognized that this embedding creates a new language. In particular, the Ada Board, speaking through the Director of the Ada Joint Program Office in a letter to the American SQL standards committee (ANSI X3H2) in April of 1987 [Castor 87], recommended that an embedding of SQL into Ada not proceed to standardization. They noted the following:

> The major objection to the embedding is that an Ada program with SQL statements embedded within it is no longer conformant with the Ada standard. ... [T]he strong typing of the Ada language is subverted by embedding SQL in Ada ... . Consequently, ... reliability of the ... Ada software using the embedding is significantly impaired. ... *The binding of Ada and SQL should provide the full capabilities of both languages.* (Italics added.)

Figure 1-2: The Pre-Compilation Process

1.3.2. All-Ada Bindings

There have been a number of proposals made which accomplish an Ada SQL interface entirely within Ada [Brykczynski1 86] [Lock 90] [Baer 90] [Rosen 90]. These proposals are very different and resemble each other only in that they use Ada and the Ada compiler as the only tools with which to implement the interface. Thus, they are all implementations of the process in Figure 1-2, using the Ada compiler as the implementation of the dashed-outline box in that figure. There is no need for a detailed review of each of these all-Ada proposals. A few selected items will serve to give the flavor of the area.

Most (but not all) of the all-Ada proposals have as a high level goal the imposition of Ada-like typing on the DBMS. Some proposals go so far as to require the interface to allow objects of any Ada type, including variant records and unconstrained arrays, in the database [Whitaker 87]. SQL supports only the storage of scalar types and fixed length strings. The proposal in [Whitaker 87] also requires that operations, e.g., string comparisons, have their Ada meaning wherever they appear. The implementation described in [Brykczynski1 86] of the ideas in [Whitaker 87] does not attempt to support database storage of complex or varying length objects. Further, the definition of string comparison found in [Brykczynski1 86] is the SQL definition, not the Ada definition.

These deviations from the philosophy of [Whitaker 87] are unavoidable. The central goal of any Ada SQL interface design is to enable Ada applications to use COTS software; SQL is just a means to that end. That implies that the SQL statement, no matter how encoded by the programmer, will be executed by the COTS DBMS, and it is impossible for the interface to affect that interpretation. (If use of COTS were not the issue, it would certainly be possible to build a DBMS satisfying the requirements of [Whitaker 87].)

The all-Ada solutions generally resort to tricks to accomplish their objectives entirely within Ada. For example, the language in [Brykczynski1 86], which is designed to resemble SQL, must spell **select** without the ‘t’ and **all** with three ‘l’s to avoid the use of an Ada reserved word. The proposal of [Baer 90] encodes the table definition given in standard SQL by:

```
CREATE TABLE Parts
    Name CHARACTER (16) NOT NULL UNIQUE
    Number CHARACTER (8) NOT NULL UNIQUE
    Description CHARACTER(240)
```

in the following Ada record type declaration:

```
type PARTS_ROW is
    record
        NAME        : NAME_COLUMN.NOT_NULL := UNIQUE;
        NUMBER      : NUMBER_COLUMN.NOT_NULL := UNIQUE;
        DESCRIPTION : DESCRIPTION_COLUMN.WITH_NULL;
    end record;
```

(Note: The complete declaration of the table parts in [Baer 90] requires 4 package instantiations—one for each column and one for the table—and six other type declarations.) This encoding succeeds at giving the database columns application-oriented types, NAME_COLUMN instead of CHARACTER (16), for example, by abusing an available Ada construct, the record type former, so that it can also play the role of an SQL table declaration. The semantics of the Parts_Row declaration is not that given it by Ada. It does not define an Ada record type with certain component names and types and with given initializer expressions. It serves only to express in compilable but distorted Ada what can already be described in perfectly straightforward SQL. (Note: The SAMeDL data description language (DDL) is the SQL DDL with small alterations. See Section 3.3.5.)

The implementations of the all-Ada interfaces generally have performance problems. The approach in [Brykczynski1 86] makes every application program depend upon the entire database definition, thereby causing excessive re-compilation as the database schema evolves. (The "fix" for this problem is an Application Scanner [Brykczynski3 88], 600 pages of Ada code.) The system described in [Lock 90] eliminates the run time excesses of [Brykczynski1 86] by increasing the length of the edit-compile-test loop. The code written by the user of [Lock 90] is compiled and executed; the output of that execution is Ada code which is compiled and then used by the application code for DBMS services. The proposal in [Baer 90] resolves that problem by postulating an "SQL DML generator" which translates compilable (but unnatural) Ada code into compilable SQL.

The fundamental objection to SQL interfaces done entirely in Ada is simply that Ada is not well suited to play the role of a database description and manipulation language. It is a tribute to the designers of Ada that such interfaces can be attempted. But the fact that it can be done is not evidence that it should be done.

1.3.3. Modularity, The SAME and The SAMeDL

It must be recognized that SQL and Ada rest on very different paradigms of software problem solving. Ada SQL program construction should be treated as an example of multiparadigm programming, a method by which software engineers "build systems using as many paradigms as [they] need, each paradigm handling those aspects of the system for which it is best suited" [Zave 89]. All-Ada solutions deliberately obfuscate the distinction between the programming language and database language paradigms; embedded solutions simply ignore it, leaving the task of reconciling the two paradigms to the programmer. The SAME and the SAMeDL treat the Ada and SQL paradigms as equal partners in application construction. The guiding principle is that application logic is written in Ada, database interactions are written in SQL, and the purpose of the interface is to reconcile the two paradigms. The key implementation concept behind the SAME is *modularity*, the idea that no piece of program text should contain both application logic and database logic, no matter how encoded. The SAME's concept of modularity is based directly on the concept of modularity to be found in the SQL standard. (See Chapter 7 of [ansiinteg 89].)

The process of Figure 1-2 is non-modular. The process assumes a single program containing both application logic, written in a programming language, and database logic, written in a data sublanguage encoded in some fashion. This style of program construction is based upon the software engineering practice of the late sixties and early seventies, the period in which DBMS first gained wide acceptance in the data processing community. DBMS applications were much smaller then and the implementation language of choice was COBOL, a language which did not provide for separate compilation. Therefore, applications were written as monolithic compilation units; there was such a thing as *the* database program. This is not the case today.

The software engineering process to which Ada is addressed "is becoming ever more decentralized and distributed. Consequently, the ability to assemble a program from independently produced software components [was] a central idea in [the] design [of Ada]." [ada 83], 1.3(4). The process is partially modeled in Figure 1-3. There is no such thing as the database program in that figure. Rather, there is a collection of independently written, separately compilable software modules, some of which implement the database services needed by the application. All production quality, "real world" Ada applications exhibit structure resembling Figure 1-3. An Ada DBMS application will modularize the database interaction in some way. This modularity, isolation or encapsulation of the text describing the database interaction has the following benefits:

- Maintenance and porting costs of such modularized DBMS applications are greatly reduced. The great bulk of the code in a modularized application is application logic written in Ada. That code is protected from any change in the DBMS.
- Modularization of the application allows for specialization of the application development team. The role of application database specialist is created. The role is not characterized simply by knowledge of SQL. The database specialist's expertise lies in understanding the meaning of the data as stored in the application database. This allows the remaining application team to concentrate on the application logic.
- Modularity raises the application database interaction to a design object. This makes database evolution smoother, as there are no hidden application dependencies buried by programmers.

Any of the other approaches can be and have been used in a modular way. But if modularity is *the* architecture of choice for DBMS applications, then modularity should be directly supported by the interface language.

Figure 1-3: Modular Program Architectures

1.3.4. Rating the Approaches

This section compares the three solution styles, embedded SQL, the all-Ada approaches and the SAMeDL, according to the requirements and issues given in Section 1.2.

- **Portability:** The issue is the extensions to standard SQL all commercial DBMS make. By definition, the SQL standard cannot and the all-Ada approaches do not deal with that issue. The two approaches are identical in that regard. The SAMeDL recognizes this problem and makes provision for these extensions. Ensuring that every non-standard feature is marked allows the effort involved in porting a SAMeDL text from one DBMS to another to be more easily calculated.
- **Shared data - Interoperability:** The all-Ada approaches generally restrict interoperability of Ada and non-Ada applications on a shared database. They specify their own, non-SQL-like data description language and they allow data types not understandable to non-Ada programs (e.g., enumerations). The SAMeDL also has its own DDL, but it has been designed to be as much like SQL DDL as possible. The SAMeDL allows user specification of enumeration encodings, making such data sharing with non Ada programs more robust.
- **Efficiency:** Embedded SQL is inexpensive in compile and run time resources. The SAMeDL compile time resource consumption is much the same as that of an embedded SQL pre-compiler. The SAMeDL introduces some run time overhead (an **if** statement, to check for nullness, for each field) and some run time space (for SQL operations in Ada programs). This overhead is usually dwarfed by the time and space overhead of running a general purpose DBMS. The all-Ada approaches are generally expensive in both compile and run time resources.
- **Range of acceptance:** As described, embedded SQL was rejected by the Ada community as represented by the Ada board and the Ada Joint Program Office. The all-Ada proposals have received no support from the DBMS community.
- **Atomic types:** The SQL standard interface for Ada supports neither enumeration nor decimal data types [Dewar 89]. The all-Ada proposals generally support enumeration, but not decimal arithmetic. In both cases, data is handled by SQL rules in the DBMS and Ada rules in the application. Data in a SAMeDL application can be operated on by the same set of rules in the DBMS and the application program, namely by the SQL rules. The SAMeDL supports enumerations in databases and can support decimal data, date-time data, or any other data class suitable for the application. (See Section 5.2.)
- **Missing information:** The SQL standard supplies null values to application programs as two parameters: the parameter itself and an "indicator parameter"; the value is null if the indicator parameter value is negative. The proposal in [Brykczynski1 86] follows this usage, which is inherently unsafe; it is very easy for programs to misinterpret or fail to examine the indicator parameter, misinterpreting the data. Both [Baer 90] and [Lock 90] and SAMeDL encapsulate the parameter and the nullness indicator into an abstract type for which nullness cannot be ignored.
- **Strong, application typing:** Standard SQL is weakly typed; it will not allow arithmetic on character strings, but is otherwise permissive. It has no type formers and, therefore, does not support application typing. Most all-Ada solutions have strong, application typing as a central goal, as does the SAMeDL.

- **Exceptional conditions:** SQL signals exceptional conditions through a status parameter. The standard does not specify the encoding of error conditions as status parameter values. The status parameters are easy to misinterpret or ignore. The all-Ada solutions tend to encode database exceptional conditions as specific, pre-defined (i.e., not user defined) Ada exceptions. [Brykczynski1 86] declares three exceptions for dealing with three particular conditions, one of which is end-of-table. It is unclear what it does with hard I/O errors, for example. [Baer 90] follows much the same course. The SAMeDL, along with [Lock 90], supports the concept of a "status map," which translates the DBMS-defined status parameter values to application-defined status parameter values (for a status parameter over a user-defined enumeration type) for whichever exceptional conditions the application expects, e.g., end-of-table. (A status map may also cause user defined exceptions to be raised.) Conditions from which the application cannot recover cause a predefined exception to be raised. This ensures that failure conditions will not be ignored or lost.

1.4. Why a New Language?

The SAMeDL was designed to facilitate the construction of Ada DBMS applications constructed in conformance with the SAME guidelines [Graham 89, Moore 89]. Those guidelines require application-oriented typing at the application interface, but they do not require strong, application typing of the SQL statements themselves. The decision to create a new language, i.e., to abandon the goal of using only standard Ada and standard SQL, was not made in order to support strong typing in SQL. Rather, the decision to support such typing was made as a consequence of the decision that standard SQL could not be used unaltered. This section will elucidate the reasoning that led to that decision.

We should be clear on what is meant by the idea of using standard SQL in the construction of SAME conformant applications. The idea is that, from a standard SQL module, it may be possible to derive the package specifications and bodies of the abstract modules defined by the SAME. The abstract module specifications differ from the specifications defined by SQL primarily in the parameter profiles; in [Dewar 89], the SQL procedures have parameters whose types are restricted to the primitive types defined by SQL, whereas the abstract module procedures have application-oriented abstract types. The problem then is to add sufficient information to the SQL Module Language to determine the types of the parameters in the interface to the Ada application, yet retaining SQL Module Language conformance. This seemingly impossible task can, in fact, be achieved through the use of structured comments.

Structured comments are a device which has found some acceptance in the Ada community for adding information to an Ada program without violating Ada language rules. The trick is used by Byron and [Luckham 87], among others. The idea is to encode information needed by a tool for generating Abstract Module procedures into a comment. It is easiest to explain through an example.

```
Module Example_Module
Language Ada
Authorization Public

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

Procedure Part_City_Open
   Input_Status Int
   SQLCODE;
   Open Part_City;

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

Procedure Part_City_Close
   SQLCODE;
   Close Part_City;
```

Figure 1-4: An SQL Module

```
with Example_Definitions; use Example_Definitions;
package Example_Interface is

type Part_Nbr_City_Pairs is record
    Part_Number : Part_Number_Not_Null;
    City : City_Type;
end record;

     -- All of these procedures may raise SQL_Database_Error

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

procedure Part_City_Fetch (
                     Part_Cities : in out Part_Nbr_City_Pairs;
                     Is_Found : out boolean);
              -- returns the relation created by open
              -- Found becomes False at end of table

procedure Part_City_Close;
              -- clean up procedure
end Example_Interface;
```

Figure 1-5: Specification of Ada Interface for Module in Figure 1-4

Figure 1-4 contains a simple SQL module containing a cursor declaration and the procedures—open, fetch and close—necessary to its use. Figure 1-5 contains the corresponding Ada package specification. Both figures are taken from [Graham 89] and reference the Parts-Supplier database also used as the basis for the examples in the SAMeDL manual [Date 75] [Chastek 90]. (The Ada package Example_Definitions is a domain package in the terminology of [Graham 89], and represents a definitional module in the terminology of [Chastek 90], containing the type definitions needed to declare the procedure parameters.) The question is: what information not present in the SQL module is needed to produce the Ada package?

Among the missing information are the types of the parameters to the procedures. Figure 1-6 displays an annotated version of the procedures in the SQL module. The **--&** token flags the remaining line segment as a comment, from SQL's point of view, and as a type name (or status map name) from the SAME's point of view.[2]

```
Procedure Part_City_Open
   Input_Status Int         --& Status_Not_Null
   SQLCODE;
   Open Part_City;

Procedure Part_City_Fetch
   Part_Number Char(5)      --& Part_Number_Not_Null
   City Char(15)            --& City_Type
   City_Indic Smallint
   SQLCODE;
   Fetch Part_City into Part_Number, City INDICATOR City_Indic
                            --& Status: Standard_Map
   ;

Procedure Part_City_Close
   SQLCODE;
   Close Part_City;
```

Figure 1-6: Annotated Module

An obvious feature of the annotated module of Figure 1-6 is the amount of redundant information it contains. Once Input_Status has been assigned the Ada type Status_Not_Null, its SQL type, Int, should be deducible. Requiring its presence is tantamount to making the module writer take an exam: there is only one right answer. The same holds for the Part_Number and City parameters. All three procedures must declare an SQLCODE parameter, although the parameter has no effect on the Ada specification. And when null values may be encountered, as for City values, the INDICATOR device for signaling nullness requires another parameter declaration and use which have no effect on the Ada specification. The module of Figure 1-6 is cumbersome and error prone. The module of Figure

[2]The annotation does not show the origins of the record type name, Part_Nbr_City_Pairs, nor the name of the parameter of that type, Part_Cities in Figure 1-5. These could be supplied in a similar way.

1-4 encoded as a SAMeDL module appears in Figure 1-7.[3] That module assumes an environment that has recorded the SAMeDL domain of each database column, much as an SQL environment records the SQL type of each database column. Only the input parameter needs to be explicitly typed by the module writer.

```
with Example_Definitions;
abstract module Example_Module is
authorization Public
    cursor Part_City
            (Input_Status : Status Not Null)
        for
            select SP.PNO Not Null named Part_Number, S.City
                from SP, S
                where SP.SNO = S.SNO
                and S.Status >= Input_Status;
        is
            procedure Part_City_Fetch is
                    fetch into Part_Cities : new Part_Nbr_City_Pairs
                    status Standard_Map;
    end Part_City;
```

Figure 1-7: The Module in SAMeDL

The most significant difference between the modules of Figures 1-4 and 1-7 is the packaging. The SELECT-FROM-WHERE block that defines the cursor has hardly been touched.[4] That is indicative of the SAMeDL design philosophy.

1.5. Fundamental Concepts of the SAMeDL

The primary design goal for the SAMeDL is the partial automation of the creation of Ada DBMS applications conforming to the SAME architecture of [Graham 89]. In particular, the requirements for the SAMeDL include support for:

- Modular program construction; separate compilation.
- Application oriented, strong typing.
    - Within the description of the database interaction (i.e., within the SAMeDL); *and*
    - At the application interface.

[3]The SAMeDL module does not generate the Ada package exactly. It generates an Ada package Example_Module containing a subpackage Part_City which in turn contains the declaration of a record type, Part_Nbr_City_Pairs, and three procedures named Open, Fetch, and Close. The procedure Part_City_Open in Figure 1-5 has become Part_City.Open.

[4]It has been necessary to make minor adjustments. The **not null** phrase gives the abstract module composer control of type names in the Ada interface; the **named** phrase gives him/her control of the parameter names. See Section 2.2. Not illustrated in Figure 1-7 are domain conversions, introduced to allow for mixed mode operations such as the product of rate and time.

- A safe treatment of missing information (null values); safe in the sense that missing information cannot be mistaken for real information.
- Robust status parameter handling.
- User data class extensibility (decimal, date, enumeration, etc.).
- Non-standard, vendor-specific extensions within the language framework.

1.5.1. Design Principles

A few simple design principles guided the choice of constructs and delineation of detail in the SAMeDL. These principles were:

1. **The principle of least invention:** Make minimal changes to SQL. Do not invent any rule or construct when an existing rule or construct can be borrowed from Ada or SQL. Do not improve on either Ada or SQL.
2. **Design for readability and maintainability.**
3. **Maintain internal consistency.**
4. **Provide ease of use but do not sacrifice user control.**

These principles are often in conflict. Their application in a given context is a matter of judgment. The SQL Data Description Language (DDL) is a good case in point. The need is to add one piece of information to the description of each database column, namely the identity of the domain for the column. Therefore, a `domain_reference` {<column definition>} is added to the end of an SQL. Other parts of column declarations are untouched, as suggested by principle 1.

In SQL, a schema is declared by a construct illustrated by:

```
create schema <name>
    . . .list of table definitions. . .
;
```

where a table definition looks like:

```
create table <name>
    . . .list of table elements. . .
;
```

In short, a schema is a variable length list of variable length lists, each list terminated by a semicolon. This structure was judged not to meet principle 2. So the decision was made that any variable length list should terminate with an explicit **end**`name`. Then, following principle 3, and considering the definitions of modules, cursors, procedures, domains etc., table definitions were changed to their final form, roughly:

```
table name is
   . . .list of table elements. . .
end [name];
```

A strong argument can be made that `name` should not be optional in an **end** delimiter. It was left optional in accordance with the "no improvement" aspect of principle 1 and the treatment of **end**s by Ada. (See Section 3.3.1.)

Principle 4 lies behind features such as the automatic provision of open, fetch and close procedures for any cursor. It is assumed that these default procedures will satisfy most application needs. When the defaults do not fit, the user may specify the procedures directly.

1.5.2. Meaning of a SAMeDL Text

The SAMeDL is a language for specifying abstract interfaces. An abstract interface is a collection of Ada declarations through which an Ada program can access the DBMS. Therefore, the meaning of a SAMeDL text is given by a translation into an Ada text, an SQL text, or both an Ada and an SQL text along with the relationship between them. Figure 1-7 gives an overview. As illustrated, a SAMeDL text may contain some data description and it may also rely on previously processed data description. The meaning of a SAMeDL text may include Ada type and/or subprogram declarations. The actions of the subprograms include, nominally, calls to procedures defined in the SQL module language. The meaning of a SAMeDL procedure includes its Ada declaration, an SQL declaration, and the definitions of the input and output parameters of the procedures declared.

Figure 1-8: The Meaning of a SAMeDL Text

1.5.3. The Three Kinds of Modules

SAMeDL statements are aggregated into text units called modules. These are similar to Ada's package aggregation but differ in two essential ways. First, SAMeDL modules cannot be nested; second, SAMeDL modules come in three different *kinds*, namely **definition**, **schema** and **abstract**. (The need for module kinds is explained later in this section.) The kind of module restricts the kind of statement that may appear within it. Module nesting was prohibited primarily because there seemed to be no useful purpose for it and it is a difficult feature to implement.

The purpose of each module kind is given in the following list.

- **Definition** modules contain the definitions of all the data which *may be shared by applications.* A definition module produces an Ada package but no SQL text.
- **Abstract** modules contain the declarations of SQL procedures *specific to an application*. These modules may also contain any of the definitions that may appear in definition modules, but those definitions will be visible only within the module. An abstract module produces both Ada and SQL texts.
- **Schema** modules contain the data definition language (DDL) of the SAMeDL, i.e., definitions of tables, views, privileges and constraints. The syntax and the semantics of these definitions are taken almost entirely from the SQL standard [ansiinteg 89]. From the SAMeDL's perspective, the interesting information is the assignment of a domain to each database column.

The distinctions made by this list clearly indicate that the tripartite division is natural. If the division were not supported by the language, i.e., if only one module kind, allowing any construct, were defined, inappropriate and unnecessary interdependencies among applications might arise. As the language is defined, interoperating applications depend only on the definitions of the data they share and not on each other's database interactions.

2. Name Space Control and Identifier De-Reference

The author of a SAMeDL module has certain external obligations to fulfill. Chief among these are the database interactions provided in the abstract interface. The names of things, in the interface or in the database, form part of those obligations. The names of database things, schemas, tables and columns are, from the module writer's point of view, predetermined and immutable. The names of procedures, parameters, types and so on that form the interface to the Ada application may be negotiated with the application programmers. Once they have been determined, however, these names, too, form part of the module author's external obligations.

This section describes the facilities available to the module author for meeting these obligations. It also describes the rules for visibility and dereferencing of names within the SAMeDL module itself. Section 2.1 describes mechanisms for setting names in the Ada interface derived from a SAMeDL text. Section 2.2 describes SAMeDL context clauses, used like Ada context clauses to import names from other modules. Section 2.3 discusses identifier de-reference in SAMeDL modules.

2.1. Specifying the Names in the Ada Interface

Most of the names in the Ada text derived from a SAMeDL text are determined in obvious ways. For example, the name of an Ada package, constant, exception, enumeration type, enumeration literal or procedure is the name of the module, constant, exception, enumeration, enumeration literal or procedure from which it is derived. The names of types, packages or other items declared in the Ada text derived from a domain declaration are determined by the declaration of the base domain. See Section 3.2.4 for more about the declaration and use of base domains.

The rules for determining *parameter names* are slightly more complex. There is generally a default available for each of these names and a mechanism for the explicit specification of any one of them. The rules and mechanisms do differ slightly depending on the class of the parameter.

- The default name of an Ada parameter derived from a SAMeDL *input parameter* is the name of the parameter. That default may be overridden using the **named** attribute. (See [Chastek 90], Section 5.6.)

  As is discussed in Section 2.3, input parameter names may conflict with column names. These conflicts can be eliminated by renaming the input parameter. The **named** attribute may be used to set the name of the parameter in the Ada interface irrespective of any conflicts.

- The default name of a *row record component* derived from a *select parameter* is the select parameter itself, *provided* that the select parameter is a simple name. If the select parameter contains any operators, including the selector dot ('.'), then there is no default component name and one *must* be supplied using the **named** attribute. (See [Chastek 90], Section 5.7.) Attempts to provide default names for slightly more complex expressions break down quickly. It was therefore deemed preferable not to make the attempt.

- The default name of a *row record component* derived from an *insert_column_specification* is the name of the corresponding column. This may be overridden using the **named** attribute. As SQL column names are limited in length, it may be desirable to give the Ada component a more meaningful, self-documenting name. (See [Chastek 90], Section 5.8.)
- *Row record parameters* have two sources of default names. The procedure taking a row record may specify a user written record declaration as effectively giving the *type* of the row record parameter. (See [Chastek 90], Section 5.9.) That record declaration may supply a default parameter name in a **named** phrase. (See [Chastek 90], Section 4.1.6.) If it does not, or if the procedure does not specify such a record declaration, then the default is *Row*. Whatever the default, it may be overridden with an **into** (or **from**) clause. Section 4.2 contains more on row record declarations, when to use them and how to name them.
- Like row record parameters, *status parameters* have two sources of default names. The status map referenced in the status clause that attaches the status parameter to the procedure may specify a default for the parameter name using the **named** phrase. (See [Chastek 90], Section 4.1.9.) If it does not, the default name is *Status*. In either case, the status clause may use the **named** phrase to override the default. (See [Chastek 90], Section 5.13.)

The *type* of a row record component or input parameter is one of the types defined by the domain of the parameter or component. The `not null` phrase, borrowed from SQL, is the means by which the module specifier selects one of the types so defined.

The default name of the row record parameter type is `Row_Type`. The into and from clauses may override the default. The name of the status parameter type is always specified by the status map referenced by the status clause.

2.2. Names Within a Module: Context Clause

The SAMeDL adopts the Ada context clause as a mechanism for importing names into modules. The use of **use** clauses is restricted to context clauses and may only reference definitional modules. SQL has no notion resembling **use**; any schema used in a module other than the schema named in the authorization clause ([ansiinteg 89], 7.1) must be explicitly referenced. This treatment of schemas is carried directly into the SAMeDL.

Unlike Ada, SAMeDL modules are not considered to be nested within any larger unit. Therefore, all externally defined names, even those of the predefined base domains in SAMeDL_Standard and constants in SAMeDL_System, must be explicitly brought into scope if they are to be used. The predefined modules are not special cases of the module construct.

As there is more than one kind of SAMeDL module, there is more than one kind of context clause. A **with** clause is used to reference a definition module; a **with schema** clause is used for schema modules. Abstract modules cannot be referenced by other modules, as they contain declarations specific to an application and are not to be shared.[5] This makes it possible for a schema module to have the same name as a definition module. This name clash must be resolved if the modules are to be used together. The resolution is accomplished by an **as** phrase.

The **as** phrase is a renaming construct which, unlike the Ada **renames**, hides the original name of the module or table.[6] Whereas the **named** attribute serves to specify a name for a declaration within the derived Ada code, and has no effect on the name space of the module itself, the **as** phrase affects only the module's name space and removes, as well as adds, a name in that space.

As a consequence of the definitions in [Chastek 90], Section 3.2, the context clause:

```
with A as B; with B as A;
```

is legal, although perverse. It sets B as the name of the module A and A as the name of the module B. The operand of the **with** is always a definition module, so the B in the second clause above is the definition module B, *not* the alias referring to the definition module A. The situation for renaming within **with schema** statements is analogous.

2.3. Identifier Dereferencing in SAMeDL

As elsewhere in the SAMeDL, the rules for identifier dereferencing attempt to accommodate both Ada and SQL rules. Both languages recognize selection as an operation on identifiers and both use the period mark as its symbol. As there is no module nesting in SAMeDL, there is an upper bound on the number of selectors or dots that may appear in any identifier.[7] SQL names never have more than two dots; in an SQL identifier of the form S.T.C, S is a schema, T a table and C a column.

---

[5]Although abstract modules cannot be referenced and shared by other modules, they can be shared or reused by Ada programs.

[6]The token **as** appears, optionally, in the from_clause of the SAMeDL between a table name and its correlation name. It does not appear at that location in SQL [ansiinteg 89], 5.20. It was added to the SAMeDL for internal consistency.

[7]The number is three. The only legal interpretation of a SAMeDL identifier of the form A.B.C.D. is as a reference within a cursor update procedure named C within a cursor named B in a module named A to the input parameter named D.

When a parameter and a column have the same name,[8] SQL disambiguates in favor of the parameter, since the column can be referenced using the table name as a prefix. The SAMeDL never automatically disambiguates an ambiguous reference. To access the parameter in this situation, a prefix denoting the procedure or cursor must be used.

The two languages differ most markedly in their willingness to use context to dereference an identifier. Up to overloading, Ada will not use context to determine a referent for a name. (SAMeDL allows overloading of enumeration literals only.) SQL, on the other hand, does use the context of an identifier to dereference it. This can produce situations which are inconceivable in Ada. Suppose that a schema named `X` contains a table named `X` which contains a column named `X`. Then, within a **from** clause, the identifiers `X` and `X.X` both reference the table; as select parameters, the identifiers `X`, `X.X` and `X.X.X` all reference the column. As these names are part of the module programmer's external obligations, the SQL treatment of these names is preserved in the SAMeDL. It is for that reason that the treatment of references in [Chastek 90], Section 3.4, distinguishes "defining locations" from "reference locations" and treats an identifier prefix as a whole, rather than as component wise.

The order of the rules in [Chastek 90], Section 3.4, implements a nesting of declarations or "hiding" discipline essentially that of Ada, but simplified by the lack of user-specified nesting. Roughly speaking, parameters and columns in the tables in scope at a given location hide all other declarations with the same simple name and also hide each other. Items (other than cursors or procedures) declared within a module hide any declaration with the same simple name in any module in context. (Cursor and procedure names are meaningful only within the text of the cursor or procedure.) Declarations with the same name in distinct modules hide each other. If an item is hidden, it cannot be referenced by its simple name alone; a prefix must be supplied. It may be that there is no such prefix and that the hidden item cannot be referenced in any way. Consider the following SAMeDL text:

```
definition module Duplicate is
    constant A is ...
end Duplicate;

with Duplicate;
abstract module Abs is
    procedure Duplicate (A : Some_Domain) is
        . . .
            . . . Duplicate.A . . .
        . . .
end Abs;
```

The identifier `Duplicate.A` in the procedure `Duplicate` references the input parameter.

---

[8] The ANSI SQL standardization committee, X3H2, modified the SQL standard in January of 1990 (see [SQLIB1 90]) by adding a colon to the front of parameter names. The use of a module language parameter in the SQL module language now has the same lexical form as the use of a variable in embedded SQL. This removes the ambiguity between column and parameter names. The SAMeDL does not use colons with parameter names.

Were there no such parameter, the identifier would be invalid. (See [Chastek 90], Section 3.4, prefix rule 2.a.i, full name rule 1.c.) The constant A in the module `Duplicate` cannot be referenced from the procedure `Duplicate`. This is very nearly the situation in Ada, except that the constant A could be referenced as `Standard.Duplicate.A` were the SAMeDL modules Ada packages. On the other hand, unlike anything in Ada, visibility in the procedure `Duplicate` to the constant A can be achieved through a context clause such as **with** `Duplicate` **as** `Dupl`; in that case, the constant A could be referenced as `Dupl.A` in the procedure `Duplicate`. On the other hand, consider the following SAMeDL example:

```
definition module Duplicate is
    constant A : Some_Domain is ...
end Duplicate;

with Duplicate;
abstract module Abs is
    procedure Proc (Duplicate : Some_Domain) is
        . . .
            Duplicate.A = Duplicate
        . . .
end Abs;
```

The equality predicate in this example is legal. The `Duplicate` on the left is a prefix, which cannot reference an input parameter; the `Duplicate` on the right can reference only the input parameter. This example shows the price paid for allowing some of SQL's eagerness to dereference into the SAMeDL.

3. Data Definition

The objective of data definition in the SAMeDL is to provide the following services to Ada SQL applications:

- A robust treatment of SQL data that effectively prevents use of null values as though they were not null while requiring no run time conversion of non-null data.
- An extended database description using abstract, application-oriented types and the application of a strong typing discipline to SQL statements.

This support is provided by the SAMeDL user-defined typing and the SAMeDL declarations.

3.1. SAMeDL User Defined Typing Support

Strong, application-oriented typing was a fundamental design goal of the SAMeDL; the typing scheme should allow for types such as Employee_Number and Part_Number instead of Integer. The SAMeDL provides this support via domains.

A domain serves the same purpose in the SAMeDL as an Ada type does in Ada; that is, a domain can be associated with an object in the SAMeDL to restrict the values that the object may assume and to define the operations that may be performed upon that object.

A domain is derived from a base domain in much the same manner that an Ada type is derived from a base type. A base domain is an encapsulation of the information needed to support a class of domains. Specifically, a base domain encapsulates:

- The Ada *types* and *operations* necessary to represent and manipulate database objects as they are realized in the Ada application.
- The SQL data type, called the *dbms type*, to be used with SQL parameters at the database interface [Graham 89].
- The base domain *parameters* which supply information needed to derive a domain from the base domain.
- The *data class* which governs the use of literals with objects of domains derived from the base domain.

Domains support the strong typing facilities of the SAMeDL. (See Section 4.1.) This section is primarily concerned with the effects of domains on the Ada application.

3.1.1. Safe Treatment of Nulls

Database objects may take a null value, indicating missing or incomplete information. However, Ada has no primitive notion of a null value.

SQL provides a set of arithmetic and comparison operators which handle incomplete information. These operators are used by SQL for its own processing, but are not exported to the application programming language. Instead, primitive data objects in SQL application programs have two parts which are not closely coupled in any strong sense: a variable holding the value of the item, and an indicator that indicates whether or not the value is null, that is, whether the value is meaningful. Programming errors in the handling of these indicators can lead to subtle software errors: seemingly valid results which are wrong.

However, an Ada abstract type can ensure the safe treatment of null values by the Ada application program by extending the class of integers, or reals, or character strings, etc., with one new value, the null. It is apparent that the two parts of an SQL value, the value and the indicator, should be incorporated into this abstract type. Further, users must be prevented from using the value without first checking the indicator.

Consider, for example, the type SQL_Int as such an extension of the class of integers. SQL_Int may be defined as:

```
type SQL_Int_Not_Null is new SQL_Standard.Int;

type SQL_Int is limited private;[9]
. . .
private
  type SQL_Int is record
    Is_Null: boolean := true;
    Value: SQL_Int_Not_Null;
  end record;
```

The SAMeDL is designed to present an application-oriented typing structure to the application programmer. Within the Ada application, values do not have type SQL_Int, but rather types like Weight_Type, Employee_Number_Type, etc., types derived from SQL_Int, so as to have the operations defined for SQL_Int. The operations that one needs to associate with the abstract type depend upon how objects of that type might typically be used.

One approach, called the minimalist (see [Graham 89]), views such an object merely as a value repository. The processing of that object, say of Weight_Type, involves a test for nullness, and if the object is not null, an extraction of its value into a different object which is guaranteed not to be null. The type of the extracted object is not Weight_Type, which may assume a null value, but of an associated type Weight_Not_Null, which does not allow null values. Thus, two Ada types are required: the abstract type (Weight_Type in the above example) which represents possibly null objects, and another type (Weight_Not_Null) which represents the not null objects associated with the abstract type's value field.

Weight_Type may be derived from SQL_Int:

```
type Weight_Type is new SQL_Int;
```

The Value field of Weight_Type, however, is not the desired type Weight_Not_Null, but rather SQL_Int_Not_Null. Thus, the extraction of a nonnull object from an object of Weight_Type must be done via a function that returns an object of type Weight_Not_Null, and not directly via record selection. Therefore, SQL_Int must be a private type.

---

[9] See the end of this section for an explanation.

The minimalist approach requires testing functions that determine if an object of the abstract type is null, and conversion functions between objects of the abstract type and its associated value type. This approach cannot be supported by a simple Ada type. However, an abstraction that binds the possibly null abstract type, called the null-bearing type, and its associated value type, called the not null-bearing type, and the above mentioned functions and operations can provide the needed support in the SAMeDL; these encapsulations are called domains.

Previously, SQL_Int was declared to be limited private. Consider the following three objects:

```
SQL_Int'(Is_Null => true, Value => 1);
SQL_Int'(Is_Null => true, Value => 1);
SQL_Int'(Is_Null => true, Value => 2);
```

If SQL_Int were not limited as well as private, an Ada application could test equality of these objects using predefined equality. The first pair of objects are equal in that sense and each is unequal to the third. But all three represent "the" null value. It is at the very least disturbing that the third object above is distinguished from the other two solely on the basis of its value field, even though its interpretation has no value.

It would seem that this difficulty can be circumvented by using a discriminated record type, using the nullness indicator as the discriminant.

```
type SQL_Int_2 (Is_Null : boolean := true) is record
  case Is_Null is
    when true =>
      null;
    when false =>
      Value : SQL_Standard.Int;
  end case;
end record;
```

Notice that the previous reasoning, that this type must be private, still holds. The Value component of objects of any type, such as Weight_Type, derived from this type will have Ada type SQL_Int_Not_Null. An extraction function returning an object of the correct type, Weight_Not_Null, is still needed. However, it would seem that this type need not be limited. Consider the following objects of type SQL_Int_2:

```
SQL_Int_2'(Is_null => true);
SQL_Int_2'(Is_null => true);
SQL_Int_2'(Is_null => false, Value => 1);
```

The earlier inconsistency has disappeared. In this representation, null values do not have value fields, so the earlier problem doesn't arise. But, suppose the first of the two null objects is the value of the salary field of a record with name field "Bush" and the second is the value of the salary field of a record with name field "Quayle". If any two null objects are equal, which is the case for Ada predefined equality on objects of type SQL_Int_2, do "Bush" and "Quayle" have the same salary? Probably not. Therefore, predefined equality is wrong for the null bearing types within domains, and those types will be limited, no matter how they are constructed.

3.2. Standard Support

Predefined (i.e., supplied by the language) support for base domains and domains is supplied by the Ada package SQL_Standard, the SAME standard support packages, and the standard base domains contained in the definitional module SAMeDL_Standard. SQL_Standard contains the Ada type declarations that define the SQL data types to the Ada compiler. For each of the SQL data types in SQL_Standard, there is a corresponding SAME standard support package and standard base domain. The support package contains the Ada type, procedure, and function declarations needed by the corresponding standard base domain.

Note that the support packages, such as SQL_Int_Pkg, and the other packages listed in the Guideline and the SAMeDL manual are only suggestions [Graham 89] [Chastek 90]. Users of the SAME and the SAME description language are free to use other support packages, provided only that their treatment of missing information is safe.

3.2.1. SQL_Standard

The fundamental feature of any Ada SQL interface is the ability to move data across the interface in a "reasonable" way. In [Engle 87], reasonable is defined as *direct*: every value stored in the database can be represented as the value of some application object *without run time data conversion*. What is needed is a set of Ada type declarations which exactly describe the SQL data types. These declarations appear in the Ada package template SQL_Standard, shown in Figure 3-1. This package is now part of the ANSI SQL standard as amended [Dewar 89].

The unspecified values in the SQL_Standard template are filled in with values specific to the DBMS. If the DBMS considers integers to be 32 bit quantities, then the declaration of SQL_Standard.Int is:

```
type Int is range -2**31 .. 2**31 -1;
```

The SQL_Standard character string data type, Char, is defined as a one dimensional array of elements from an implementation defined character set. If that character set is seven bit ASCII, this is captured by

```
package Character_Set renames Standard;
subtype Character_Type is Character_Set.Character;
type Char is array (positive range <>) of Character_Type;
```

If Char is to be based upon some other character set (EBCIDIC, national character sets, for example), then suppose there is a package named Host_Strings_Pkg that contains the Ada character enumeration type Host_Character, that describes that character set. Then Ada DBMS applications can use that set by coding this part of SQL_Standard as:

```
package Character_Set renames Host_Strings_Pkg;
subtype Character_Type is Character_Set.Host_Character;
type Char is array (positive range <>) of Character_Type;
```

```
package SQL_Standard is
  package Character_Set renames csp;
  subtype Character_Type is Character_Set.cst;
  type Char is array (positive range <>) of Character_Type;
  type Smallint is range bs..ts;
  type Int is range bi..ti;
  type Real is digits dr;
  type Double_Precision is digits dd;
  type Sqlcode_Type is range bsc..tsc;

  subtype Sql_Error is Sqlcode_Type
    range Sqlcode_Type'FIRST .. -1;
  subtype Not_Found is Sqlcode_Type
    range 100..100;
  subtype Indicator_Type is t;

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

end SQL_Standard;
```

Figure 3-1: The Package SQL_Standard

Ada DBMS applications that are not sensitive to the details or collating sequence of character representations can be ported between character sets without modification.

3.2.2. Standard Base Domains

The standard base domains are the base domains supplied by the language in the predefined definitional module SAMeDL_Standard; they are a "start up" kit for the SAMeDL. The SAMeDL standard base domains are based on the types from SQL_Standard and model the types supplied in Ada's package Standard. They differ from the Ada types in package Standard in that the base domains are not domains, whereas the Ada types in package Standard are types.

The standard base domains are SQL_Int, SQL_SmallInt, SQL_Char, SQL_Real, SQL_Double_Precision, SQL_Enumeration_as_Char, and SQL_Enumeration_as_Int. Notice that there are two standard base domains for enumerations: SQL_Enumeration_as_Char, and SQL_Enumeration_as_Int. SQL_Enumeration_as_Char has a DBMS type of character, that allows users of domains derived from SQL_Enumeration_as_Char to represent an Ada enumeration in the database by character literals, while SQL_Enumeration_as_Int has a DBMS type of integer, allowing an integer representation in the database (See Section 3.2.5 for a further explanation.)

Note that new base domains may be declared to provide support for types not in the standard base domains (See Section 5.2.)

3.2.3. Using the SAME Standard Base Domains and Support Packages

Consider a Weight domain whose non-null values are non-negative integers. Such a domain can be declared in a SAMeDL **definition** or **abstract** module[10] by the syntax:

```
domain Weight is new SAMeDL_Standard.SQL_Int
    (First => 0, Last  => SAMeDL_System.Max_SQL_Int);
```

The domain pattern for the base domain SQL_Int (see Appendix C.1 of the Reference Manual [Chastek 90]) causes the following Ada code to appear in the specification of the Ada package corresponding to the module containing that definition. (See Section 5.2 for a discussion of patterns in base domain declarations.)

```
type Weight_Not_Null is new SQL_Int_Pkg.SQL_Int_Not_Null
    range 0 .. SAMeDL_System.Max_SQL_Int;

type Weight_Type is new SQL_Int_Pkg.SQL_Int_Type;

package Weight_Ops is new SQL_Int_Pkg.SQL_Int_Ops
    (Weight_Type, Weight_Not_Null);
```

The type Weight_Not_Null is an Ada integer type, with the Ada predefined integer operations. This section describes: the procedures, functions and operations defined on the so-called null bearing type, Weight_Type; how those procedures, etc., were generated; and the thinking that lead to the generation of those subprograms in that way.

As described in Section 3.1.1, we will need at the very least a function to test objects of Weight_Type for null and to extract the value of non-null objects. The support package SQL_Int_Pkg contains the function specification

```
function Is_Null (Value : SQL_Int) return boolean;
```

Therefore a function Is_Null taking a Value of Weight_Type is generated by the rules of Ada type derivation. This is the null testing function.

The value extraction function presents a slightly more difficult problem. Its parameter profile involves both of the types in the Weight domain. This function cannot be derived from any function in any support package.[11] Therefore, the function Without_Null, taking an object of type Weight_Type and returning an object of type Weight_Not_Null is generated in the package Weight_Ops. (Weight_Ops also generates a function With_Null, which generates a Weight_Type object from a Weight_Not_Null object. This encapsulation function is useful when inserting data into the database.)

The following functions, procedures and operations defined on Weight_Type are generated

---

[10]A **with** clause for SAMeDL_Standard and one for SAMeDL System are needed in the context of the module.

[11]The type derivation rules will generate a function taking an SQL_Int object and returning a Weight_Not_Null object; and they will generate a function taking a Weight_Type object and returning an SQL_Int_Not_Null object. But the rules of Ada in [ada 83], Section 3.4, will not generate the "cross product" function.

by the subprogram derivation rules from functions, procedures and operations defined in SQL_Int.

- The function Null_SQL_Int, returning an object of type Weight_Type with the property that Is_Null(Null_SQL_Int)=true. Null_SQL_Int is a function, rather than a constant, precisely so that it will derive in this way.
- Overloadings of the four arithmetic operators and the operators **mod**, **rem** and **. These operators take and return Weight_Type objects. Their meaning is given by the following piece of pseudo-code.
- ```
  if either operand is null, then return Null_SQL_Int;
      else return the result of applying the operator to the operands;
      end if;
  ```
- Overloadings of the comparison operators other than = (and /=), and the functions Equals and Not_Equals. These take operands of type Weight_Type and return a result of type Boolean_with_Unknown defined in SQL_Boolean_Pkg.[12] (See [Chastek 90], Appendix C.)
- The meaning of these functions is given by the following piece of pseudo-code.
- ```
  if either operand is null, then return UNKNOWN
      else return the result of applying the operator to the operands;
      end if;
  ```
- Overloadings of the comparison operators including = (and /=). These take operands of type Weight_Type and return a result of the predefined Ada type Standard.boolean. The meaning of these functions is given by the following piece of pseudo-code.
- ```
  if either operand is null, then return false;
      else return the result of applying the operator to the operands;
      end if;
  ```

The overloaded comparison and arithmetic operators allow programmers to manipulate database values directly, without first extracting them into non null bearing objects. The operators are defined to imitate the SQL semantics of null value. There are two sets of overloadings for the comparison operators. The second set implements a conservative view of missing information: a given condition is false unless it can be proven to be true.

As was discussed in Section 3.1.1, the null bearing types within domains are **limited** in order to disallow Ada predefined equality for objects of those types. However, neither do **limited** objects have a predefined assignment operator, and one must be defined. Regrettably, Ada does not allow ":=" to be overloaded. A procedure Assign, having operands of type Weight_Type, is needed. For character string objects, objects of domains based on the SQL_Char standard base domain, the Assign procedure is generated by derivation. The numeric standard base domains generate their assignment procedures in the generic subpackage of each standard support package. This was done to facilitate range checking of

[12]The rules of Ada require that any overloading of "=" return a boolean result. (See [ada 83], Section 6.7.)
  ```

null bearing types. The Assign procedure for Weight_Type objects will enforce the range constraint defined for Weight_Not_Null. In other words, the procedure call:

```
Assign (Weight_Object, With_Null(-1));
```

will raise Constraint_Error. (The exception is raised by Assign, not by the conversion function With_Null.)

The support packages are designed to make maximum use of sub program derivation and minimal use of generic instantiation. Alternatively, it is possible to achieve much the same result using only generic instantiation. For example:

```
generic
  type Without_Null_Type is range <>;
package Int_Support_Pkg is
  type SQL_Data (Is_Null : boolean := true) is limited private;

  function Without_Null (Left : SQL_Data)
                        return Without_Null_Type;
  function "+" (Left, Right : SQL_Data)
                return SQL_Data;
. . .
private
  type SQL_Data (Is_Null : boolean := true) is record
    case Is_Null is
      when true => null;
      when false =>
        Value : Without_Null_Type;
    end case;
  end record;
end Int_Support_Pkg;
```

In this case the Weight domain is declared by:

```
type Weight_Not_Null is range 1 .. 10,000;
package Weight_Pkg is new Int_Support_Pkg (Weight_Not_Null);
```

Potentially null weights are values of the Ada type Weight_Pkg.SQL_Data. All of the operations on that type are generated by the instantiation. For most compilers, this will result in very large application executables.

3.2.4. Base Domains

The previous sections described the standard support for data definition. Such support is not sufficient when a new data type such as *datetime* is required by a user, or when the operations supplied by the standard support package are not what is needed. In such cases, a SAMeDL user may define a new base domain and create an associated support package.

Recall that a base domain is the abstraction of a class of domains, based upon an abstract (null-bearing) type, the corresponding not null-bearing type, and the support package containing the functions and operations required to support these types. It is a template for defining domains, providing the support needed to create and use domains. This support includes:

- The specification of the parameters, such as range constraints, needed to create the domain from the given base domain.
- The generation of Ada code, generally type declarations and package instantiations, to support the domain in Ada.
- Instructions needed by the SAMeDL processor, such as the names of the null and not null-bearing types.

For a further discussion of the creation of new base domains, see Section 5.2.

3.2.5. Enumeration Domains and Maps

Enumeration domains are a special subclass of domains. They consist conceptually of a list of enumeration literals, together with a mapping that specifies how the enumeration literals are to be represented in the database. The list is specified by the enumeration parameter_association, while the mapping is specified by the map parameter_association. The map is necessary as SQL does not support enumerated types. An enumeration base domain contains a default map for enumeration domains derived from it. The default map may be overridden by specifying a map when an enumeration domain is declared, thus providing support for existing enumerations whose values already exist in database, or for data to be used by another language when the default map is not appropriate.

For example, the standard base domain SQL_Enumeration_as_Int contains the default map **pos**.[13] Suppose that an Ada application is to use the enumeration:

```
type weekday is (Sun, Mon, Tue, Wed, Thu, Fri, Sat);
```

to describe an existing database column which represents Sunday by 1, Monday by 2, etc. The default map **pos**, which represents Sunday by 0, Monday by 1, etc., cannot be used. A new domain Weekdays, however, can be derived from SQL_Enumeration_as_Int by overriding the default map **pos**:

```
domain Weekdays is new SQL_Enumeration_as_Int
  (enumeration => weekday,
   map         =>
     (Sun => 1,
      Mon => 2,
      Tue => 3,
      Wed => 4,
      Thu => 5,
      Fri => 6,
      Sat => 7));
```

Enumeration literals may be used in SAMeDL statements in the natural way. When the

[13]The map **pos** simulates the Ada predefined attribute function 'POS by mapping an enumeration literal to its position (i.e., a nonnegative integer) in its enumeration declaration.

statement is translated into SQL, the enumeration literal is replaced by its database value, as given by the map. Notice that an enumeration domain has two orderings, one implied by the order of the enumeration literals in the enumeration literal list, and a second ordering implied by database literals within the enumeration's map. For the sake of consistency, these ordering should be the same; otherwise, comparisons in the SAMeDL statement may give unwanted results.

3.3. SAMeDL Declarations

Ada and SQL are fundamentally different languages, as is reflected in the syntax each uses for declarations. In general, the SAMeDL's syntax for declarations attempts to use SQL-like syntax for SQL declarations, and Ada-like syntax for Ada declarations. This is an application of the principle of least invention and the principle of design for readability and maintainability. (See Section 1.5.1 for an explanation of the tension between those principles as it applies to SAMeDL declarations.)

Most SAMeDL declarations, such as modules and base domains, adhere to a general format, as discussed in the following section. Other SAMeDL declarations, called scalar declarations, associate domains with objects (e.g., column definitions and select parameters). Their format is discussed in Section 3.3.2.

3.3.1. Declaration Format

A SAMeDL declaration is either a module, base domain specification, base domain body, domain, subdomain, cursor, procedure, cursor procedure, constant, record, enumeration, exception, status map, table, privilege, or view declaration. (See [Chastek 90], Sections 3.2, 4.1, 5.1, 4.2, 4.2.1, and 4.2.2.) The syntax of a declaration is exactly that of SQL whenever that declaration has little or no effect upon the generated Ada, environment, or interface. For example, the syntax of unique constraint and privilege definitions are exactly as they appear in SQL. Constant, enumeration, record, status, and exception declarations, are generally from Ada and have no SQL parallel; hence, they adopt an Ada-like syntax. These constructs were added to support the readability and maintainability of the SAMeDL.

The following ordered list summarizes the general form of SAMeDL declarations:

- Optional keyword **extended**, when applicable.
- Keyword that identifies the declaration (e.g., module, record, constant, etc.).
- Name of the declaration.
- Parameter list, if applicable.
- Keyword **is**.
- Body of the declaration.
- Keyword **end**.
- Optional repetition of the name of the declaration.
- Closing semi-colon.

When any part of the above list (excluding the optional keyword **extended**) does not naturally apply to a particular type of declaration, that part, and all subsequent parts, are omitted. For example, a domain declaration requires a reference to a base domain and an optional parameter association list. (See [Chastek 90], Section 4.1.4.) As there is no declaration body, the **end** and optional repetition of the declaration name are also omitted. Exception declarations simply declare that an identifier is an exception, with no other information necessary. Thus, for exceptions, only the type and the name of the declaration are present.

Note that cursor declarations are something of an exception to the above rules, in that the cursor's query and order by clause appear immediately after the cursor's parameter list, and before any cursor procedures appearing in the body of the cursor.

3.3.2. Scalar Declarations

A SAMeDL scalar declaration is the special subclass of SAMeDL declarations that associate a domain with the object declared. These are constant declarations, record component declarations, column definitions, and parameter declarations. (See [Chastek 90], Sections 4.1.5, 4.1.6, 4.2.1, and 5.6.) These declarations are of the form:

- Name of the declaration, where applicable.
- Optional **named** phrase.
- Domain reference.
- Optional keywords **not null**.

Adherence to this format is loose. Column definitions maintain a strong SQL flavor while parameter declarations adopt an Ada-like style. (See [Chastek 90], Section 5.6.) However, certain general rules apply to the **named** phrase, and the combination of the domain reference and the keywords **not null**. (See Section 3.3.3 for a discussion of the **named** phrase.)

The Ada type of the object declared by a scalar declaration is determined by the domain reference, together with the optional **not null**. If **not null** appears in the declaration, or if the domain referenced in the declaration does not support null values, then the object's Ada type is the not null-bearing type from that domain; otherwise, the Ada type is the corresponding null-bearing type.

3.3.3. Named Phrases

The name of an object declared in a SAMeDL text is usually preserved in the corresponding Ada text. For example, the SAMeDL record declaration:

```
record Parts is
  The_Number : Part_Number_Domain;
  The_Name   : Name_Domain;
end Parts ;
```

generates the Ada record type declaration:

```
type Parts is record
  The_Number : Part_Number_Type;
  The_Name   : Name_Type;
end Parts;
```

This is the case for packages (corresponding to modules), subpackages (corresponding to cursors), procedures, types declared for enumerations and constants.[14]

The names of procedure parameters and, for row record parameters, the name of the row record type and the names of its components, can be specifically set by the SAMeDL programmer by use of the optional **named** phrase. This allows the SAMeDL programmer to avoid name clashes as might occur if an input parameter name clashed with a column name. For example, if table P contains a column named The_Part, then the SAMeDL procedure declaration:

```
procedure Delete_Parts
  (The_Part named Part_To_Delete : Part_Number)
  is
delete from P
  where PNAME = The_Part
;
```

generates the Ada procedure specification

```
procedure Delete_Parts (Part_To_Delete : in Part_Number_Type);
```

Note that the parameter name in the Ada procedure specification is Part_To_Delete, not The_Part.

For row record parameters, the default names of the components are given by the corresponding select parameters. A select parameter in the SAMeDL is a value expression (see Section 4.1), followed by the optional **named** phrase and not null phrase. (See [Chastek 90], Section 5.7.) The named phrase allows the SAMeDL programmer to assign a meaningful name to a select parameter and thereby name its corresponding row record component. (See Section 4.2.) The named phrase is necessary when a value expression has no obvious associated name; it may be desirable when the default name of the value expression is not meaningful to the Ada application.

Consider an Ada application that needs to retrieve an employee’s total compensation (i.e., the sum of the employee’s salary and benefits) based upon that employee’s employee number. If the database table S contains columns EMPNO (employee number), EMPSAL (employee salary), and EMPBENS (total dollar value of the employee’s benefits), then EMPSAL+EMPBENS is the desired sum. Neither EMPSAL nor EMPBENS accurately describes total compensation. There is no way for the SAMeDL to infer a meaningful name, as it might were the column EMPCOMP (defined as total employee compensation) in the table

[14] Types declared by domain instantiations are named by the base domain patterns. (See [Chastek 90], Section 4.1.4.)

S. Thus, the SAMeDL programmer must supply a meaningful name for such "complex" select parameters via the named phrase.

In the SAMeDL:

```
procedure Employee_Comp (Employee_Number : Emp_Num_Domain) is
  select EMPNO          named Employee_Number,
         EMPSAL+EMPBENS named Total_Comp
  from S
  where EMPNO = Employee_Number;
end ;
```

The generated Ada is:

```
type Row is record
  Employee_Number : Employee_Number_Type;
  Total_Comp      : Comp_Type;
end record;

procedure Employee_Comp (Employee_Number :     Emp_Number_Type;
                         Row             : out Row_Type);
```

In the above example, EMPNO is the default name for the first select parameter. The Ada application, however, may not consider EMPNO a meaningful or desirable name: the **named** phrase allows the SAMeDL programmer to assign a more meaningful name, in this case Employee_Number. Named phrases also serve this purpose in insert column specifications. (See [Chastek 90], Section 5.8.)

3.3.4. Constants

Constants are a convenience that allows the SAMeDL user to assign a name, and optionally a domain, to a frequently used static expression. There is no notion of a constant declaration in SQL. Thus, the SAMeDL constant declaration is modeled after the Ada constant declaration, with the domain reference in the SAMeDL constant declaration modeling the Ada type indication in an Ada constant declaration. As in Ada, if the static expression in the SAMeDL constant declaration is not numeric, a domain reference must be present.

An Ada constant declaration is generated from each SAMeDL constant declaration. That Ada constant will have the value of the SAMeDL constant declaration's static expression, as determined by the rules of SQL, not Ada. (See [Chastek 90], Section 4.1.5.) This provides for one semantics for value expressions in the SAMeDL; whether a value expression appears in a select parameter list or as a static expression in a constant declaration, its value is determined by the rules of SQL. (See Section 4.1.) This also provides the Ada application with access to the same value seen by the database.

In Ada, a constant of a limited type may be declared only within the program unit that declares that limited type. The null-bearing type of domain is typically limited (see Section 3.1.1), and declared in the support package. The Ada constant generated from a SAMeDL constant declaration will be declared in the package corresponding to the SAMeDL module that contains the constant declaration, not in the support package. Thus, the generated Ada constant must not be of the associated domain's null-bearing type. If that not null-bearing type is also limited, the SAMeDL constant declaration will be marked as erroneous.

3.3.5. Schema Modules

Schema modules make table names, as well as column names and their associated domains, known to the environment. Further, any of these declarations may contain extended elements, and need to be so marked. The SQL syntax for schemas, tables, columns, etc., is not sufficient for those purposes. Thus, the general syntax adopted for schema modules and table definitions is that of the SAMeDL, rather than that of SQL. Other schema features, such as the default clause, unique constraint, and referential constraint definition have no effect on the environment, and therefore appear exactly as they do in SQL.

Notice that a column definition contains an optional SQL data type. This is the type of the column in the SQL table definition. When not specified, the SQL data type is assumed to be the referenced domain's DBMS type. When specified, the SQL data type and the referenced domain's DBMS type must be interconvertible in both directions. This is required, as the domain's DBMS type defines any associated parameter's type at the concrete interface, and the parameter may be used in connection with an insert or a fetch statement. The SQL data type and the domain's DBMS type, however, need not be the same; SQL allows conversions in both directions between certain data types, such as between single and double precision.

The SQL data type need not be specified. However, the SQL data type should be included for verification purposes if the associated table already exists in the database, or if there is no promotion of type (e.g., from single to double precision).

4. Data Manipulation

This chapter describes certain aspects of data manipulation that are defined in the SAMeDL. It is not concerned with data manipulation defined by SQL. There is no discussion of SQL verbs. Rather, the chapter contains sections on value expressions, row records, into and insert-from clauses, insert values lists, the differences between SQL and SAMeDL cursor declarations, and SAMeDL standard post processing.

4.1. Value Expressions: Formation and Use

Three principles underlay the design of value expressions. They are:

1. Formation rules for value expressions should be a mixture of Ada and SQL rules; no new rules should be invented, if possible.
2. The formation rules should enforce a strong typing discipline.
3. The formation rules should be the same everywhere within the SAMeDL; value expressions within constant declarations are written in the same language and interpreted in the same way as value expressions in select parameters, atomic predicates, or anywhere else.

A value expression language has four sets of rules defining and interpreting the legal expressions. Those sets are:

1. Lexical rules, describing the tokens of the language.
2. Concrete (or context free) grammar rules, describing the language without reference to types.
3. Abstract (or context sensitive) grammar rules, describing the typing rules.
4. Semantic rules, describing the function denoted by the expression.

Both Ada and SQL define each of these pieces and agree on none of them. The problem is to choose rules of each language appropriately such that the result is easily remembered and intuitively clear. It is not possible to choose freely among the rule subsets of the two languages; they are not mutually independent.

Observe that there is no choice in defining the semantics of SAMeDL expressions when the expressions appear within SQL contexts, i.e., as select parameters or operands of atomic predicates. Expression semantics *must* be those of SQL, as the expressions will be evaluated by the SQL engine using the rules of SQL and it is impossible to modify that behavior. It seems natural, therefore, to use SQL concrete grammar since that grammar establishes operator precedence and therefore affects the meaning of the expression.[15]

[15]Ada and SQL concrete grammars differ in that SQL has fewer operators than Ada, dropping **abs, mod, rem**, and **, and the languages treat unary operators (+, -) in different ways. The only known consequent of this difference is that SQL will accept the expression x--1 (== x+1), whereas Ada requires x-(-1). SQL will parse -x*2 as (-x)*2 and Ada will parse it as -(x*2), but these are equivalent as arithmetic expressions.

The strong typing goal suggests that we use Ada's abstract syntax, since that is where typing rules are applied. There are choices to be made in applying Ada typing rules to SQL statements, as will be shown.

This leaves lexical issues. We would like to choose either the Ada or the SQL lexical rules, but nothing said so far forces a choice. Salient features of Ada and SQL lexical rules are given in the following list:

1. Both Ada and SQL are case insensitive languages.[16] They define the same set of identifiers (sequences of letters, underscores and digits, starting with a letter and not ending with an underscore) except that (1) each language reserves a different set of identifiers; (2) SQL identifiers are limited to 18 characters in length.
2. Every Ada numeric literal (which is not an Ada based_literal) is also an SQL numeric literal, but not conversely. SQL allows 1. and 1.0; .1 and 0.1; etc.
3. Ada uses the double quote as a string delimiter; SQL uses the single quote.

Given that the SAMeDL is based on the SQL module language, the choice was made to use SQL lexical rules for string and numeric literals. A SAMeDL identifier that names a schema, table or column must necessarily be an SQL identifier; otherwise it is an Ada identifier. In summary, the design of the SAMeDL value expression language makes the following choices:

1. Literals are formed using SQL rules; identifiers are either SQL identifiers or Ada identifiers, depending on what they identify.
2. The content free, concrete grammar of expressions is that of SQL, which is a minor variant of the Ada concrete expression grammar.
3. The context sensitive, abstract grammar of expressions, the typing rules, are those of Ada.
4. The meaning of a value expression is the meaning assigned to it by SQL.

Serious obstacles confront the design of a value expression language that satisfies these rules, particularly rules 3 and 4. What exactly are the "typing rules" of Ada? We may say that an expression satisfies the typing rules of Ada if each operator in the expression has an interpretation consistent with the usage of the operator in the expression. Some of those interpretations may be user-defined, some will be predefined. The SAMeDL does not have a means for users to define operator parameter profiles; therefore, the SAMeDL rules allow just those expressions which use only the so-called predefined operators of Ada, ([ada 83], Section 4.5 (6)). An operator is predefined for a SAMeDL domain if that operator is predefined for Ada types of the same data class as the domain.

There is a subtle, but easily explained, divergence from this interpretation of rule 3 in the SAMeDL. Ada predefines multiplication and division of any fixed point type and the

---

[16] Ada is specifically stated to be case insensitive [ada 83], 2.3(3). SQL allows only upper case letters in identifiers [ansiinteg 89], 5.3.

predefined type Standard.Integer [ada 83], 4.5.5(7). The SAMeDL has no domain that can play the role of Standard.Integer. Therefore, the SAMeDL allows multiplication and division of any fixed point type by integer literals or named numbers, effectively replacing the predefined type Standard.Integer with the anonymous domain, *universal_integer*.

Ada also predefines multiplication and division of any two fixed point types; therefore, SAMeDL does so as well. The results are treated differently in the two languages for reasons that stem from the languages' treatment of type conversions, more properly called domain conversions in the SAMeDL.

Type conversions are essential in any strongly typed language, but SQL is not strongly typed and has no such operations. When defining the SQL semantics of a SAMeDL domain conversion, there is no alternative to the choice made by the SAMeDL, which is to erase the conversion. (See [Chastek 90], Section 5.10, SQL Semantics rule 3.) Insofar as an Ada type conversion has no semantic effect (i.e., is the identity operation on values, as when one integer type is converted to another with identical subtype constraints), this presents no difficulty. The SAMeDL can not reproduce the effects of an Ada type conversion that is not the identity, as when a real type is converted to an integer; it cannot do so because no such operation is available in SQL, and the SAMeDL operator semantics are just those of SQL.

The problem here is one of perception. The author of an expression involving a domain conversion may have a mental model of that expression that reproduces the Ada model. For example, suppose the existence of two objects or database columns, one named Elapsed_Time, having an integer based domain, TIME, and one named Average_Speed, having the floating point domain, SPEED. The author of the expression:

    Elapsed_Time *TIME(Average_Speed)

might well believe that the product is an integer. It will, however, be a floating point value, as the SQL semantics of that expressions is simply

    Elapsed_Time * Average_Speed

which is a floating point result. In order to ensure that the author of these expressions thinks about what they mean, a warning message must be issued for any conversion which would not be the identity in Ada. (See [Chastek 90], Section 5.10, Recursive Cases, rule 2.a.i.) These conversions are generally ones that lose accuracy, as when a floating point number is converted to an integer.

We are now in a position to explain the difference in the Ada and SAMeDL treatment of fixed multiplication and division. In both Ada and SAMeDL, the result of such an operation has the anonymous type *universal_fixed*. The only operation on that type in Ada is type conversion, which serves to specify result precision. The corresponding domain conversion in the SAMeDL would not have that effect; therefore, it seems foolish to require it. Thus, in the SAMeDL, a product or quotient of fixed objects is treated like a fixed literal.

When used in *assignment contexts*, value expressions calculate values to be assigned to some object, an Ada object for an expression as a select parameter or constant, a database column for an expression in an insert or update statement. In assignment contexts, SQL will perform a limited class of data conversions. These are precisely the cases which are allowed by the SAMeDL. (See [Chastek 90], Section 3.5.)

4.2. Using Row Records and Into and Insert-From Clauses

Data received from, or transmitted to, an SQL database is represented by a sequence of individual parameters in SQL. For example, the SQL select statement:

```
select SNO, SNAME, STATUS CITY
  into SSNO    INDICATOR NUMINDIC,
       SSNAME  INDICATOR NAMEINDIC,
       SSTATUS INDICATOR STATUSINDIC,
       SCITY   INDICATOR CITYINDIC
  from S
  where SNO = 2
```

retrieves the supplier number SNO, the name SNAME, status SSTATUS, and city SCITY from the row in table S whose supplier number SNO is 2. The Ada procedure declaration for this statement is given by:

```
with SQL_Standard;
procedure Get_Data (Supplier_Number : out Int;
                    NumIndic        : out Indicator_Type;
                    Supplier_Name   : out Char;
                    NameIndic       : out Indicator_Type;
                    Supplier_Status : out Int;
                    StatusIndic     : out Indicator_Type;
                    Supplier_City   : out Char;
                    CityIndic       : out Indicator_Type;
                    SQLCODE         : out SQLCODE_Type);
```

The Ada application, however, should see a database object which represents the information needed from the database. The Ada record which encapsulates the individual pieces of database information (e.g., the out parameters in the Get_Data procedure), and thereby represents the database object, is called a *row record*.

The most natural way to define a row record type is based on the select parameter list of the associated SAMeDL select statement. The order of the components of the row record type is given by the order of the select parameters, the name of each component by use of the **named** phrase or by default, and the type by the domain of each select parameter, in conjunction with its not_null phrase.

The into clause of a procedure's fetch or select statement, or the insert-from clause in the case of an insert-values statement, specifies the name and type of the row record parameter to be associated with that procedure. The SAMeDL processor will generate a default row record parameter name and type for use with a procedure that contains a select, fetch, or insert-values statement, if no into clause is specified in the select or fetch statement, or insert-from clause in the insert-values statement.

The into and insert-from clauses may be used to name the generated Ada row record type. For example, the SAMeDL procedure declaration:

```
procedure Part_Name (Part_Number : Part_Number_Dom) is
    select PNAME
    into Part_Name_Rec : new Part_Name_Number_Type
    from P
    where PNO = Part_Number
;
```

will generate the following Ada :

```
type Part_Name_Number_Type is record
  Part_Number : Part_Number_Type;
end record;

procedure Part_Name (Part_Number : in out Part_Number_Type);
```

The use of **new** in the SAMeDL into clause above names the generated Ada record declaration Part_Name_Number_Type.

If no into clause had been specified in the SAMeDL procedure Part_Name, the following Ada would have been generated:

```
type Row_Type is record
  Part_Number : Part_Number_Type;
end record;

procedure Part_Name (Row : in out Row_Type);
```

The SAMeDL user may supply a reference to a SAMeDL record declaration (see [Chastek 90], Section 4.1.6) in the into or insert-from clause of the associated procedure. In this case, no Ada record declaration is generated as a result of the Part_Name declaration. Note that if an into or insert-from clause contains a record reference, then the names, types, and order of the components of the referenced record must match exactly the names, types, and order of the components of the generated default record type. This requirement provides an explicit error check by the SAMeDL processor on the referenced record. As the names can be user controlled via the associated select parameter's named phrase, component names are included in this check.

SAMeDL record declarations and into and insert-from clauses provide a mechanism for the sharing of record objects in the Ada application. The application must, however, take care when using this facility. Procedures should share a row record type only when the database objects corresponding to the row record parameters in those procedures have the same meaning. This is not, and cannot be, enforced by the language.

Consider the following record and procedure declarations in which an SCODE of Y indicates an active supplier, while an SCODE of P indicates a potential supplier:

```
record Supplier_Record is
  SNO   : Number_Domain;
  SNAME : Name_Domain;
  SCITY : City_Domain;
end Supplier_Record;

procedure Supplier (The_City : City_Domain) is
  select SNO SNAME SCITY
  into Active : Supplier_Record;
  from S
  where SCITY = The_City and SCODE = ’Y’;
end Supplier;

procedure Potential_Supplier (The_City : City_Domain) is
  select SNO SNAME SCITY
  into Potential : Supplier_Record;
  from S
  where SCITY = The_City and SCODE = ’P’;
end Potential_Supplier;
```

The procedure Supplier returns database objects representing active suppliers, while the procedure Potential_Supplier returns database objects representing suppliers who could be, but are not yet currently active suppliers. The SAMeDL will allow these procedures to share a row record type, as the names, types, and order of the components of each procedures default row record type are the same. However, the Supplier_Record type does not capture the different meanings of the objects retrieved by the procedures. That is, Supplier objects represent active suppliers, while Potential_Supplier objects represent inactive, but possible, suppliers; this information is lost when Supplier_Record represents these objects. This loss of information can be eliminated by adding SCODE to the Supplier_Record and altering the procedures as follows:

```
record Supplier_Record is
  SNO   : Number_Domain;
  SNAME : Name_Domain;
  SCITY : City_Domain;
  SCODE : Code_Domain;
end Supplier_Record;

procedure Supplier (The_City : City_Domain) is
  select SNO SNAME SCITY SCODE
  into Active : Supplier_Record;
  from S
  where SCITY = The_City and SCODE = ’Y’;
end Supplier;

procedure Potential_Supplier (The_City : City_Domain) is
  select SNO SNAME SCITY SCODE
  into Potential : Supplier_Record;
  from S
  where SCITY = The_City and SCODE = ’P’;
end Potential_Supplier;
```

The new Supplier_Record can now represent the objects retrieved by both the procedures without loss of information; the procedures may share a row record type.

Defining distinct row record types for every procedure will lead to a proliferation of record type declarations, while the indiscriminate sharing of row record types can lead to information loss and possible misuse of shared objects.

4.3. Insert Values Lists

In SQL, an insert-values list may contain parameters, literals, and the keyword "null". An SQL insert statement that contains an insert-values list must also contain an insert-column list of equal length. The values in the insert-values list and the columns in the insert-column list are associated by position, and each value must be compatible with the data type of its associated column. (See [ansiinteg 89], Section 8.7.)

The SAMeDL places further restrictions upon the insert-values list in an insert-values statement. The insert-values list is restricted to contain only static values and column names, where a static value is either a literal, a constant reference, or the keyword "null".

Where SQL allows parameters, the SAMeDL requires column names. Further, the SAMeDL requires that any column name that appears in an insert-values list within an insert-values statement must be the same column name that appears in the corresponding position in that insert-values statement's insert-column list. This restriction serves as an error checking device.

Positional parameters such as those in the insert-values lists are notoriously prone to error. The use of named parameters in SAMeDL insert values lists would therefore be justified by the principle of readability and maintainability. However, further improvements were viewed as a violation of the principle of least invention. (See Section 1.5.1.)

4.4. Cursors

Cursors in the SAMeDL differ from cursors in SQL in that the SAMeDL:

- Encapsulates a cursor's procedures with the cursor declaration.
- Specifies the parameters to a cursor's query as parameters to the cursor's declaration rather than as parameters to an open procedure.
- Provides default open, close, and fetch procedures for a cursor, if those procedures are not supplied.

In SQL, the cursor declaration and the statements that refer to that declaration are bound by the cursor name; those statements may be separated from each other and from the cursor declaration by any number of other SQL statements.

In the SAMeDL, the cursor declaration textually contains all cursor procedures for that cursor to provide better readability and maintainability.

In SQL, the parameters to a procedure that contains an open statement do not appear in that statement, but rather in the cursor specification within the associated cursor declaration. The following is an incomplete example of an SQL module that declares a cursor, and an open procedure for it:

```
declare CUR cursor
  for
    select SNO SNAME
    from S
    where SNO = PARM1
;
procedure CUROPEN PARM1 INTEGER SQLCODE;
  open CUR ;
```

The procedure CUROPEN declares integer parameter PARM1, but the open statement within CUROPEN does not refer to PARM1; an SQL open statement can specify only the name of its associated cursor declaration. Notice that the cursor specification within the declaration of CUR does refer to PARM1 (i.e., in the where clause). In fact, the parameters to the open procedure are parameters to the cursor specification.

In a SAMeDL cursor declaration, the parameters to the query are associated with the cursor declaration, not with the open procedure, again to provide improved readability. For example:

```
cursor CUR ( PARM1 : integer not null)
  for
    select SNO SNAME
    from S
    where SNO = PARM1
  ;
  is
  procedure CUROPEN is
    open CUR ;
  end CUR;
```

Note that at runtime, the values of the parameters to the cursor declaration are passed to the generated Ada open procedure. Continuing the current example, the Ada code generated by the SAMeDL cursor declaration CUR is, in part :

```
package CUR is
  procedure CUROPEN (PARM1 : integer);
end CUR;
```

Finally, the SAMeDL automatically provides default open, close, and fetch procedures for any cursor. That is, if any of the open, close, or fetch statements do not appear in the list of cursor procedures for a cursor declaration, the corresponding default declaration

```
procedure open is open;

procedure close is close;

procedure fetch is fetch status : standard_map;
```

is assumed by the SAMeDL processor. These default procedures should satisfy most application needs; in the cases when these defaults are not desired, the user may specify alternative procedures directly. This is in accordance with the principle that the SAMeDL should provide ease of use without sacrificing user control. (See Section 1.5.1.)

Notice that the default cursor procedure declaration for the fetch procedure has the status map Standard_Map attached. Standard_Map signals end of table by returning **false**. (See Section 4.5 and [Dewar 89], Section 8.6.)

4.5. Standard Post Processing

Standard post processing is a central feature of the SAMeDL. Its goal is to ensure a uniform, user-specified treatment of the DBMS status parameter, SQLCODE. Errors in the treatment of SQLCODE can easily escape detection at system integration and test, and be delivered to the field, as these errors often involve values of the status parameter resulting from rare failure modes. SAMeDL standard post processing eliminates these errors.

SAMeDL standard post processing is based upon the principle that exceptional DBMS conditions can be organized into three classes, the classification based on the severity of the error and the kind of recovery possible. The classes are:

1. States that are properly part of the application logic. An application reading the tuples of a cursor will expect to reach end of file eventually. A database update application might or might not expect updates to be rejected occasionally due to potential constraint violations. States in this class are not error states.
2. States that indicate situations the application wishes to tolerate, but are not logically part of the application. If, for example, the DBMS reports it has exhausted storage resources, the application may react by cleaning up and reentering the DBMS. That logic is likely to be the same for all potential sources of the condition, e.g., for all SQL statements, and is not part of the logic of the application.
3. States that indicate situations the application need not tolerate. Failures due to syntactical errors should never occur in fielded software. If, for example, a database reorganization has dropped a table or column and invalidated an SQL statement, the application can do little to recover. At best, a rather cold restart is possible. In any case, some one must be informed of the nature of the failure so that corrective steps can be taken. Almost all of the *potential* SQLCODE values and almost none of the SQLCODE values encountered by an application will be in this class.

This classification of SQLCODE values cannot be made *a priori* for all applications. Different actions may be appropriate for different applications in the same circumstances. An update violating a DBMS integrity constraint may be expected by an application that does not, or cannot, do its own constraint checking. If that violation occurs in an application that is meant to pre-validate all updates, the error may indicate a serious logic flaw. An interactive application encountering a storage overflow problem may wish to recover gracefully in order to maintain dialogue with the user; a batch application may be willing to abort immediately. The classification mechanism must be flexible enough to account for these differences.

The status map (see [Chastek 90], Section 4.1.9) is the mechanism by which the SAMeDL allows the user to specify and utilize SQLCODE classification schemes. A status map is attached to a procedure via a status clause. (See [Chastek 90], Section 5.13.) Each procedure may therefore have its own status classification scheme. Although a DBMS may have hundreds of error codes, so that there are in principle trillions of classification schemes, in practice there are no more than handful of useful status maps.

A status map effects the classification of SQLCODE values into one of the three classes described above in the following way:

1. An SQLCODE value to which the status map assigns an enumeration literal is a value in the first class. These values are expected by the application and are handled in the normal flow of control.
2. An SQLCODE value to which the status map assigns a **raise** statement is a value in the second class. These conditions are generally of the form: storage exhausted or deadlock detected, and are about the system supporting the application rather than about the application itself. Recovery from these conditions may be possible without substantial loss; that is, a warm restart is possible.
3. SQLCODE values to which the status map makes no assignment are values in the third class. These are conditions from which recovery is essentially impossible.

When a procedure has no status map, then success ('0') is the only SQLCODE value of the first type; all other values are of the third type. If the procedure has a map, then zero is an SQLCODE value like any other. Presumably, it would be placed into the first class by the status map.

Note that the exceptions raised in cases two and three above are declared in SAMeDL modules.[17] Therefore, there is an Ada exception declaration with the same name in a generated Ada package specification. That declaration is visible to any Ada program wishing to see it. Thus, the handler for any of these exceptions can be declared at the appropriate location in the call tree and the application code can be written as though the DBMS were incapable of failing in unforeseen ways.

The purpose of the procedure Process_Database_Error, which is called in the case of an irrecoverable error, is to gather information about the cause of the error to ensure that it is not lost. Most DBMS have a facility for returning an English text string describing an error condition. Someone will need to be notified of the problem and corrective action, maintenance of the application or of the database, must be taken. Process_Database_Error is the means of giving that person the information with which to fix the problem.

[17]SQL_Database_Error is declared in SAMeDL_Standard. The other exceptions are user-declared.

5. Extensions

5.1. Introduction

Most SQL DBMS implementations offer users more power than is available in the standard. These extensions to the standard are of two different kinds: the ability to store objects of data types not covered by the standard; and operations not covered by the standard. For example, *datetime* is a frequently offered, vendor-defined data type not in the SQL standard, while outer join, connect, and disconnect are examples of nonstandard operations. SAMeDL support for these extensions is imperative, as they generally meet real user needs. The problem is, how to implement these nonstandard features (thus making them available to the Ada application) while maintaining the benefits of standardization.

The SAMeDL supports both forms of extensions. New data types can be added to the SAMeDL environment by the user (i.e., without altering the processor) by the addition of appropriately defined base domains and support packages. New operations, however, must be supplied by the vendor of the SAMeDL compiler, as they require modification of the SAMeDL processor.

5.2. Extended Data Types

To deal with data type extensions, the SAMeDL contains statements which allow the user to add base domains to the SAMeDL environment without modification to the SAMeDL processor. Once added, these act like the standard base domains (which are part of the language) in allowing users to define new domains derived from them.[18] Data types such as *datetime*, with operators reflecting the DBMS operators, may be offered to the Ada application without changing the language definition.

The declaration of a base domain has two parts: a specification and a body. The specification of the base domain SQL_Int, for example, is:

```
base domain SQL_Int is
  First : integer;
  Last  : integer;
end SQL_Int;
```

This specification states that any domain derived from this base domain may supply two integer parameters that may be referenced in the body of the base domain. Base domain bodies contain information that describes the Ada declarations (types, instantiations, etc) needed to define a domain derived from this base domain. This information takes the form of a pattern or template, into which the values of parameters (such as First and Last) may be substituted. For example, the body of SQL_Int is (in part):

---

[18]SQL_Int is a standard base domain, as is SQL_SmallInt, SQL_Real, etc. (See Section 3.2.2.)

```
base domain body SQL_Int is
  domain pattern is
    'type [Self]_Not_Null is new SQL_Int_Not_Null'
      '{ range [First] .. [Last]};'
  'type [Self]_type is new SQL_Int;'
  'package [Self]_Ops is new SQL_Int_Ops('
    'With_Null_Type => [Self]_Type,'
    'Without_Null_Type => [Self]_Not_Null);'
  for not null type name use '[Self]_not_null';
  for null type name use '[Self]_type';
  for data class use integer;
  for dbms type use integer;
```

A domain such as Weight, whose values are the integers from one to ten thousand, can be written as:

```
domain Weight is new SQL_Int (First=>1, Last=>10000);
```

The curly brackets surrounding the range constraint in the template indicate that the phrase is optional. It is generated only if both First and Last are given values. In the case of Weight, the range constraint is generated, with First replaced by 1, and Last replaced by 10000.

In addition to the parameters declared within a base domain specification, there are predefined (i.e., recognized by the SAMeDL processor) parameters. For example, in the body of SQL_Int, the predefined parameter **Self** will be replaced by "Weight" (i.e., the name of the domain being defined). **Parent** is another such predefined parameter; its value in a domain declaration is the name of the base domain or domain that the declared domain is being derived from. If **Parent** had occurred in the base domain body of SQL_Int, then in the Weight domain declaration, **Parent** would represent "SQL_Int".

More generally, base domains describe how to declare the Ada objects necessary to support domains derived from the given base domain. More specifically, the three functional needs addressed by base domains are:

1. Information needed to derive a domain from the given base domain.
2. Ada code that must be generated to support the declared domain.
3. Information needed by the SAMeDL processor to generate the procedures and parameters at the database interface.

Roughly speaking, the base domain specification addresses the first functional need, while the patterns and options of the base domain body address the second and third functional needs, respectively. In the context of the Weight example, the base domain parameters First and Last constrain the Ada type Weight_Not_Null, which is generated from the first pattern in the base domain body of SQL_Int. The last option in the base domain body of SQL_Int informs the SAMeDL processor that the SQL data type integer should be used at the database interface with an object of the domain Weight.

The SAMeDL employs the primitive, but very general, technique of string processing with simple substitution for **self**, **parent**, and base domain parameters in the processing of base domain bodies. The user must use care when specifying patterns, as the SAMeDL processor does not verify that the processed strings constitute valid Ada code. The string processing technique was adopted for several reasons. First, although the SAMeDL standard base domains represent a minimal set of base domains for the language, the standard support provided is extensive. Still, certain often used, nonstandard data types, such as *datetime*, require user-defined base domains. In practice, we expect new base domains to be added infrequently. Further, there is no need to place restrictions upon the style of the generated Ada declarations, nor to perform Ada syntactic and semantic checking within the SAMeDL processor.

There are several issues with respect to user-definable base domains that must be addressed by the SAMeDL implementor. The standard options specified in the language reference manual (see [Chastek 90], Section 4.1.2) are necessary for the definition of the SAMeDL language; in practice, other options may be necessary. For example, the SAMeDL processor may require base domain options that specify:

- Name of the Ada support package.
- Name of the assignment procedure for objects of any limited type, and the name of the package that contains its declaration.

5.3. Extended Functions and Operations

Extending the set of operations recognized by the language is more problematic. Whereas base domains can be created by the SAMeDL database administrators, extensions can be defined only by SAMeDL implementors. The extensions will generally be specific to the target DBMS and are, by their existence, violations of direct portability. Rules and suggestions regarding extensions to the SAMeDL are intended to maintain the syntactic and semantic style of the language and to minimize the potential portability problems.

Extensions can add any desired functionality to the language, but there are some restrictions that must be made. Extensions must remain strongly typed; use the standard error processing path (see Section 4.5), and follow the general form of SAMeDL procedures.

The SAMeDL grammar prohibits an extension from influencing the formation of procedure names, cursor names, module names, table names, view names, status clauses, status parameter names and types, and standard post processing. While the syntax and semantics of an extension is implementation-defined, the SAMeDL Language Reference Manual states that "any portion of an extension whose semantics may be expressed in standard SAMeDL shall be expressed in standard SAMeDL syntax." (See [Chastek 90].) For example, database data returned from extended procedures or cursors should be returned to the application in record objects. The vendor is strongly urged to use syntax and semantics in the spirit of the SAMeDL when adding extensions.

The implementor should note that an SQL implementation which employs a nonstandard SQL syntax does not require an extension to SAMeDL; the SAMeDL processor can simply generate the nonstandard SQL. Also, if an extended function or operation is addressed in SQL2[19] (see [Melton 90]), it is suggested that the implementor use SQL2 as a guide. This may ease portability problems and make extension of the an SQL-based SAMeDL processor to an SQL2-based processor easier should SQL2 be accepted as a standard.

Vendor-supplied extended data types, such as *datetime*, might include extended operations, such as ">", "=", or "+". Such extended operators must preserve the strong typing rules of the SAMeDL as they may appear in value expressions. (See Section 4.1.) For instance, operators for *datetime* ">" should not be defined when one of the operands has, for example, a user-defined type such as Weight_Type.

Extended predicates must also preserve the strong typing rules of the SAMeDL. For example, Sybase's TRANSACT-SQL (see [Darnovsky 87]) supports an outer join that combines two tables to create an new relation. Suppose that the table SUPPLIERS contains columns of SNO (supplier number), SNAME (supplier name), and SCITY (city where supplier is located), while the table PARTS contains columns PNO (part number), PNAME (part name), and PCITY (city where the part is stored). The TRANSACT-SQL statement

```
select SNO SNAME PCITY
  from SUPPLIERS, PARTS
  where SUPPLIERS.SCITY *= PARTS.PCITY
```

selects all suppliers; that is, for each row in the table SUPPLIERS, there will be a selected triple (SNO, SNAME, PCITY). If the supplier identified by SNO is located in a city where some part is stored, then the corresponding PCITY entry is that city; otherwise, the PCITY entry is null.

However a vendor may implement the SAMeDL extension for predicates such as an outer join, the operands (in the above example SUPPLIERS.SCITY and PARTS.PCITY) must obey the strict typing discipline specified by [Chastek 90], Section 5.11.

Notice that the SAMeDL allows extended input parameters with an **out** mode. Consider an application that performs multiple simultaneous transactions: each call to the database (i.e., database operation) must be seen by the database as part of a single transaction; however, operations from the multiple transactions are received by the database in an unpredictable order. A database-supplied transaction identifier, however, allows the application to bind database operations to their respective transactions. The procedure that returns the transaction identifier from the database requires an *out* parameter for the transaction identifier. The SAMeDL restricts the mode of parameters in nonextended procedures; extended procedures, however, must be permitted to have *out* parameters.

[19] SQL2 is a proposed standard to succeed SQL.

To retain control over portability costs, modules in the SAMeDL that use extensions must state their intention and point to the extension. This is done with the keyword **extended**. For example:

```
extended abstract module Example is
  extended procedure Example_Extend is
  . . .
  an extended statement may appear here.
  end Example_Extend;
end Example;
```

The reader of these modules can very quickly see whether any given modules require effort to port. For those that do, the pieces that must be examined are clearly identified.

References

[ada 83] *Reference Manual for the Ada Programming Language.*
Ada Joint Program Office, January 1983.

[ANSI2 86] *(Working Draft) Database Language SQL Addendum-2*
American National Standards Institute X3H2, 1986.

[ansiinteg 89] *Database Language - SQL with Integrity Enhancement*
American National Standards Institute, 1989.
X3.15-1989.

[Baer 90] Baer, D., Sum, K.
SQL_ArmAda An Ada-Appropriate Interface to SQL.
Technical Report CCI-90-2-4, Competence Center Informatik, May 1990.

[Boyd 87] Boyd, S.
SQL and Ada: The SQL Module Option.
Technical Report, COMPASS, Wakefield, MA, 1987.

[Brykczynski1 86] Brykczynski, B. R., Friedman, F.
Preliminary Version: Ada/SQL: A Standard, Portable, Ada-DBMS Interface.
Technical Report P-1944, Institute for Defense Analyses, Alexandria, VA, July 1986.

[Brykczynski2 87] Brykczynski, B. R.
Methods of Binding Ada to SQL: A General Discussion.
Technical Report, Institute for Defense Analyses, Alexandria, VA, 1987.

[Brykczynski3 88] Brykczynski, B. R., Friedman, F., Heatwole, K., Hilliard, F.
An Ada/SQL Application Scanner.
Technical Report M-460, Institute for Defense Analyses, Alexandria, VA, March 1988.

[Castor 87] Castor, V. L.
Letter to Donald Deutsch.
1987.

[Chastek 90] Chastek, Gary, Graham, Marc H., Zelesnik, Gregory.
The SQL Ada Module Description Language - SAMeDL.
Technical Report CMU/SEI-90-TR-26, ADA255781, Software Engineering Institute, Carnegie Mellon University, November 1990.

[Chastek2 91] Chastek, Gary J., Graham, Marc H., and Zelesnik, Gregory.
Rationale for SQAL Ada Module Description Language SAMeDL.
Technical Report CMU/SEI-91-TR-4, ADA235780, Software Engineering Institute, Carnegie Mellon University, March 1991.

[Cobol 78] COBOL Journal of Development.
Materiel Data Management Center, 1978.
Earlier versions appeared in 1973 and 1968.

[Darnovsky 87] Darnovsky, M., Bowman, J.
TRANSACT-SQL User's Guide.
Sybase, Inc., 1987.

[Date 75] Date, C. J.
An Introduction to Database Systems, First Edition.
Addison-Wesley, 1975.

[Dewar 89] Dewar, R. B. K.
Database Language - Embedded SQL X3.168-1989
American National Standards Institute, 1989.

[Donaho 87] Donaho, J. E. D., Davis, G. K.
Ada-Embedded SQL: the Options.
Ada Letters VII(3):60-72, May June 1987.

[Engle 87] Engle, C., Firth, R., Graham, Marc H., Wood, W. G.
Interfacing Ada and SQL.
Technical Report CMU/SEI-87-TR-48, ADA199634, Software Engineering Institute, Carnegie Mellon University, December 1987.

[Graham 89] Graham, Marc H.
Guidelines for the Use of the SAME.
Technical Report CMU/SEI-89-TR-16, ADA228027, Software Engineering Institute, Carnegie Mellon University, May 1989.

[Ichbiah 90] Ichbiah, J.
Public Comments.
June 1990.
ISO/WG9 meeting.

[idms 78] IDMS DML Programmer's Reference Guide.
Cullinane Corporation, 1978.

[Lock 90] *User's Manual for a Prototype Binding of ANSI-Standard SQL to Ada Supporting the SAME Methodolgy.*
Software Technology for Adaptable, Reliable Systems (STARS) Program, 1990.

[Luckham 87] Luckham, D. C.
ANNA, A Language for Annotating Ada Programs: Reference Manual.
Springer-Verlag, 1987.

[Melton 90] Melton, J., ed.
(ISO working draft) Database Language SQL2.
International Organization for Standardization and American National Standards Institute X3H2, 1990.

[Melton2 92] Melton, J., ed.
Database Language SQL2.
International Organization for Standardization, 1992.

[Moore 89] Moore, J. W.
Conformance Criteria for the SAME Approach to Binding Ada Programs to SQL
August 1989.
Special Report, CMU/SEI-89-SR-14.

[quel 76] Stonebraker, M., Wong, E., Kreps, P., and Held, G.
The Design and Implementation of INGRES.
ACM Transactions on Database Systems 1(3):189-222, September 1976.

[Rosen 90] Rosen, J. P.
Ada to SQL-Database Binding (Draft Proposal).
1990.

[SQLIB1 90] ANSI X3H2.
SQL Information Bulletin Nbr 1.
1990.

[sysr 76] Astrahan, M. M .et al.
System R: Relational Approach to Database Management.
ACM Transactions on Database Systems 1(2):97-137, June 1976.

[Vasilescu 90] Vasilescu, E. N.
Approaches in Interfacing Ada and SQL (Draft).
1990.

[Whitaker 87] Whitaker, W. A.
Requirements for an SQL Binding to Ada.
November 1987.

[Zave 89] Zave, P.
A Compositional Approach to Multiparadigm Programming.
IEEE Software 6(5):15-25, September 1989.

Table of Contents

List of Figures