N89-16371

A Database Management Capability for Ada[*]

Arvola Chan
Sy Danberg
Stephen Fox
Terry Landers
Anil Nori
John M. Smith

Computer Corporation of America
4 Cambridge Center
Cambridge, MA 02142

[*]Ada is a Registered Trademark of the U.S. Government (AJPO)

1. Introduction

The data requirements of mission-critical defense systems have been increasing dramatically. Command and control, intelligence, logistics, and even weapons systems are being required to integrate, process, and share ever increasing volumes of information. To meet this need, systems are now being specified that incorporate database management subsystems for handling storage and retrieval of information. Indeed, it is expected that a large number of the next generation of mission-critical systems will contain embedded database management systems. Since the use of Ada has been mandated for most of these systems, it is important to address the issues of providing database management capabilities that can be closely coupled with Ada.

Under sponsorship by the Naval Electronics Systems Command and the Defense Advanced Research Projects Agency, Computer Corporation of America has been investigating these issues in the context of a comprehensive distributed database management project. The key deliverables of this project are three closely related prototype systems implemented in Ada.

1. LDM (local data manager): an advanced, centralized database management system that supports a semantically rich data model designed to improve user productivity. It can be used either stand alone or as an integral part of the other two prototype systems.

2. DDM (distributed data manager): a homogeneous distributed database management system built on top of a collection of LDMs in a computer network. It supports the transparent distribution and replication of data in order to provide efficient access and high availability.

3. Multibase: a retrieval-only system that provides a uniform interface through a single query language and database schema to data in preexisting, heterogeneous, distributed databases. It utilizes LDM for managing its local workspace during the processing of a global query.

All three systems are designed to support identical interfaces for interactive use and for use through application programs written in Ada. Fundamentally, they support a "semantic" data model that captures more application semantics than conventional data models. The interactive language is called Daplex. Daplex has been designed to be an Ada compatible database sublanguage. The syntax of many of its constructs for data definition and data manipulation has been borrowed from Ada. The application programming interface is called Adaplex. It consists of an expression-level integration of Daplex's data manipulation constructs with Ada. This paper identifies a set of requirements for a modern database management capability for Ada that has driven our design for the aforementioned prototype systems. It provides an overview of the Daplex and Adaplex languages, and a summary of the functional capabilities and technical innovations we have incorporated in the LDM, DDM, and Multibase systems.

2. Requirements

Providing a database management capability for Ada is not an easy task. Our goal is to provide a complete set of modern database management capabilities which are consistent with the style and philosphy of Ada and which are well integrated with the Ada language and its support environments. This section summarizes the major requirements of a database management capability for Ada. These requirements can be grouped into three general areas: classes of databases that must be supported, operating environments, and compatibility with Ada.

Classes of databases

Ada programs will need to access three classes of databases. The first class consists of centralized databases. These databases reside at a single location and are managed by a DBMS that executes on a single computer. The second class consists of distributed databases. These databases can be fragmented, distributed, and replicated across a number of (possibly geographically separated) sites. They are managed by a DBMS that executes on a number of computers that are connected by a communications network. Distributed databases provide improvements in reliability, survivability, and expandability over centralized databases. The third class is pre-existing databases. These are databases (possibly centralized or distributed) that are managed by existing DBMSs. These DBMSs are not implemented in Ada. They provide different sets of functional capabilities and support different interface languages. An important requirement for an Ada database capability is to provide a single Ada interface to all of the above classes of databases. In other words, the particular class of database being accessed should be transparent to the Ada database application programmer.

Operating Environments

An Ada DBMS must be able to operate effectively in both an Ada programming support environment (APSE) to facilitate the development of Ada database application programs, and in an Ada run time environment to support the execution of these programs. To provide for the needs of these two environments, the DBMS must have two operating modes: shared and embedded. Shared mode is normally used in an APSE. A single copy of the DBMS supports the simultaneous development of multiple Ada database application programs in this mode. The interface between the application programs and the DBMS is a loosely-coupled one, each being executed as a separate Ada program. Thus, each application program can be changed without impacting the DBMS or other application programs. Embedded mode is typically used in a run time environment. Once the application programs have stabilized, they can be loaded together with the DBMS into a single Ada program. The applications and the DBMS then operate as separate Ada tasks that synchronize and communicate via rendezvous, thereby achieving a higher degree of interface efficiency at the expense of reduced flexibility. Embedded mode is less flexible than shared since a change to one application causes the other application and the DBMS to be relinked.

Compatibility with Ada

Ada has made a large contribution to improving program integrity through strong type checking at compile time and constraint checking at run time. It is important that an Ada DBMS provides the same degree of integrity on the Ada program data that it manages. An Ada DBMS should support all of the Ada data types, including derived types, subtypes, and type attributes. It should also support the same degree of run time constraint checking. Note that this cannot be easily (or efficiently) accomplished by simply providing an Ada interface to an existing (non-Ada) DBMS. Let us illustrate this with a simple example. Suppose an Ada programmer wants to store a set of employee records in a database. The Ada type definitions for this record may look like:

```
type YEARS is new INTEGER range 0..50;

type EMPLOYEE is
   record
      NAME              : STRING(1..30);
      YEARS_OF_SERVICE : YEARS;
      SALARY            : INTEGER;
   end record;
```

Suppose that the Ada programmer writes a program that contains a transaction that adds one to the YEARS_OF_SERVICE component of each employee record. There are two ways to process this transaction. One way is to retrieve the YEARS_OF_SERVICE component for each record in the database and return it to the application program, add one and then store it back in the database. This is a very inefficient way of processing since it results in a lot of data being sent from the DBMS to the application program and then back again. A much more efficient method is to have the DBMS perform the update directly. That is, the application program can instruct the DBMS to add one to the YEARS_OF_SERVICE component of each record. This results in no data being returned to the application program. However, the DBMS must now take the responsibility of insuring that all new values of YEARS_OF_SERVICE remain within the specified range. It is not acceptable for the DBMS to blindly change each value of YEARS_OF_SERVICE, only to have the application programs that retrieve the data at a later time discover that some values have become illegal.

3. Daplex

Data models and associated query languages have evolved significantly over the past two decades. The early hierarchical models were superseded by the network and relational models. The latter are in turn being superseded by so-called semantic data models. Our overall DBMS project is based on a semantically rich data model called Daplex which combines and extends the key features of earlier data models. For example, Daplex's modelling constructs are a strict superset of those found in the relational model. Daplex is designed to enhance the effectiveness and usability of database systems by capturing more of the meaning of an application environment than is possible with conventional data models. It describes a database in terms of the kinds of entities that exist in the application environment, the classifications and groupings of these entities, and the structural interconnections among them. The semantic knowledge captured in Daplex is not only meaningful to end

users, but is also usable by the database system and database administrator for the purposes of query and physical schema optimization. For example, knowledge of the nature of relationships between types of entities (i.e., whether they are one-to-one, many-to-one, or many-to-many) can be used to control the appropriate clustering of entities of different types that are likely to be accessed together, both in a centralized and in a distributed environment.

The basic modelling constructs in Daplex are entities and functions. Entities correspond to conceptual objects. Entities are classified into entity types, based on the generic properties they possess. Functions represent properties of conceptual objects. Each function, when applied to an entity of appropriate type, yields a single property associated with that entity. Such a property is represented by either a single value or a set of values. These values can be simple, being drawn from Ada supported scalar types and character strings, or composite, consisting of references to entities stored in the database. We illustrate these constructs with an example.

Consider a university database modelling students, instructors, departments, and courses. Figure 1 is a graphical representation of the definition of
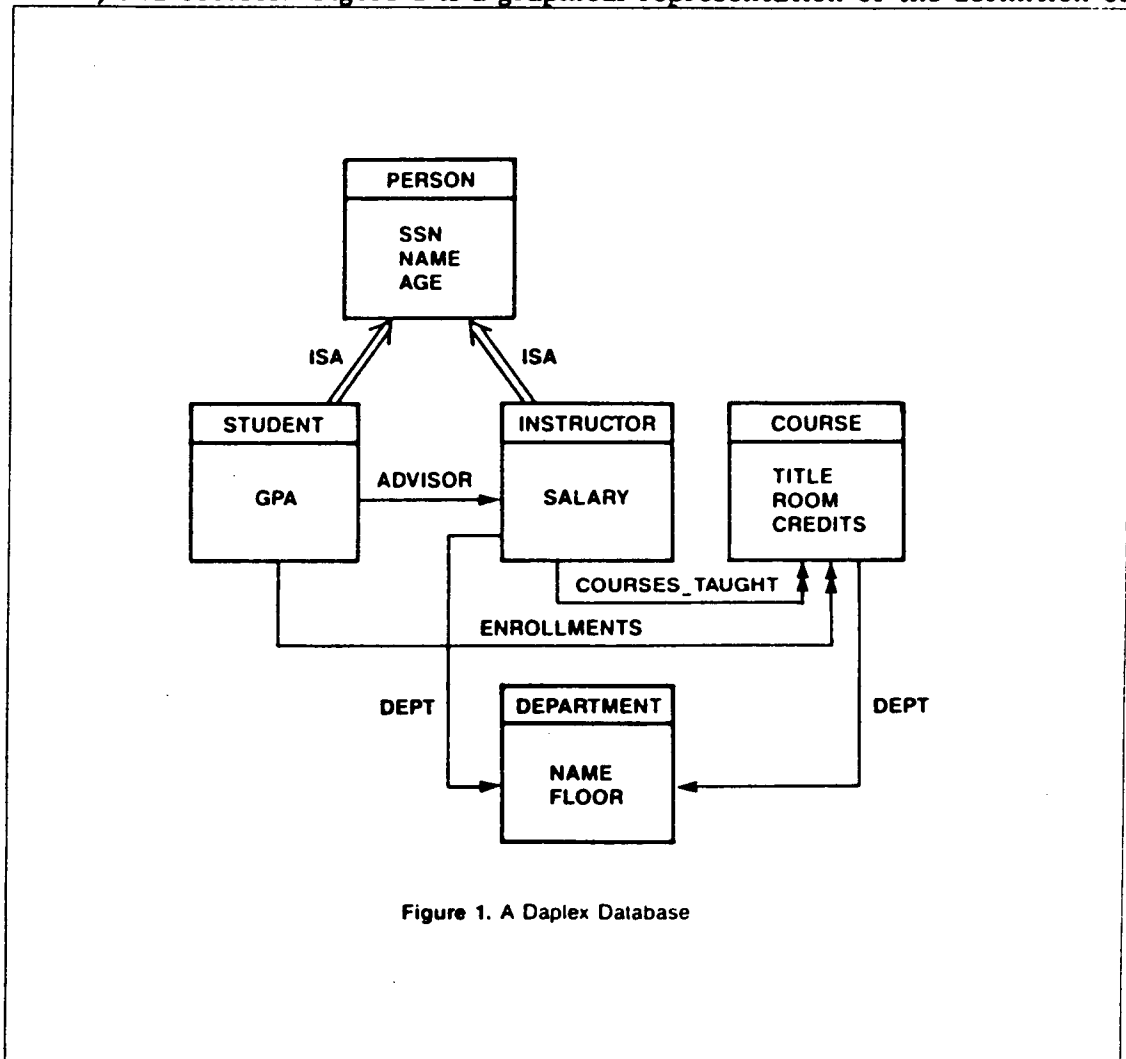

Figure 1. A Daplex Database

such a database. The rectangles depict entity types. The labels within the rectangles depict functions that range over Ada scalar and string types. The single-headed and double-headed arrows represent single-valued and set-valued

functions that map argument entity types to result types. The double-edged arrows indicate isa (subtype) relationships.

One major difference between Daplex and the relational model is that referential integrity constraints [Date81], which are extremely fundamental in database applications but not easily specifiable in a relational environment, are directly captured. For example, when a student is inserted into the database, the database system will ensure that it is assigned a valid instructor, i.e., one that is existent in the database. Likewise, when an instructor is to be removed from the database, the database system will see to it that no dangling references result, i.e., there are no more students in the database who have the instructor in question as advisor.

Another important semantic notion captured in Daplex is that of a hierarchy of overlapping entity types. In relational systems, a real-world entity that plays several roles in an application environment is typically represented by tuples in a number of relations. In the university application environment, we might have an instructor entity named John Doe and a student entity also named John Doe. In this case, it might be desirable to impose the constraint that the age of John Doe as an instructor should agree with the age of John Doe as a student. One possible strategy in a relational system is to represent this information only once by having a relation person that stores the age information, and relying on joining operations to determine the age information for students and instructors. In Daplex, we can specify that student and instructor are subtypes of person whereby we can utilize Daplex's function inheritance semantics to simplify the formulation of queries and updates. Figure 2 shows a relational equivalent of the university database. Figures 3 and 4

PERSON (SSN, NAME, AGE)

STUDENT (SSN, ADV-SSN)

INSTRUCTOR (SSN, DEPT)

COURSE (ROOM, CREDITS)

ENROLLMENTS (SSN, TITLE)

COURSES_TAUGHT (SSN, TITLE)

Figure 2. A Relational Schema

shows a Daplex query and its equivalent in SQL [DATE84]. The intent of this query is to print the names of all students taking a class held at room "F320" and taught by an instructor in the "CS" department. Notice how explicit join terms have to be introduced in the SQL query, which tend to obscure readability. On the other hand, the absence of such constructs from the Daplex query allows the query to be read in a more or less English-like manner. A complete description of the Daplex data model and access language can be found in [SLRR84].

```
for each S in STUDENT where
        "F320" is in ROOM(ENROLLMENTS(S))
        and
        DEPT(ADVISOR(S)) = CS
loop
        PRINT(NAME(S));
end loop;
```

Figure 3. A Daplex Query

```
SELECT PERSON.NAME
   FROM    PERSON, STUDENT, ENROLLMENTS, COURSE, INSTRUCTOR
   WHERE PERSON.SSN = STUDENT.SSN
        AND PERSON.SSN = ENROLLMENTS.SSN
        AND ENROLLMENTS.TITLE = COURSE.TITLE
        AND COURSE.ROOM = "F320"
        AND STUDENT.ADV-SSN = INSTRUCTOR.SSN
        AND INSTRUCTOR.DEPT = CS
```
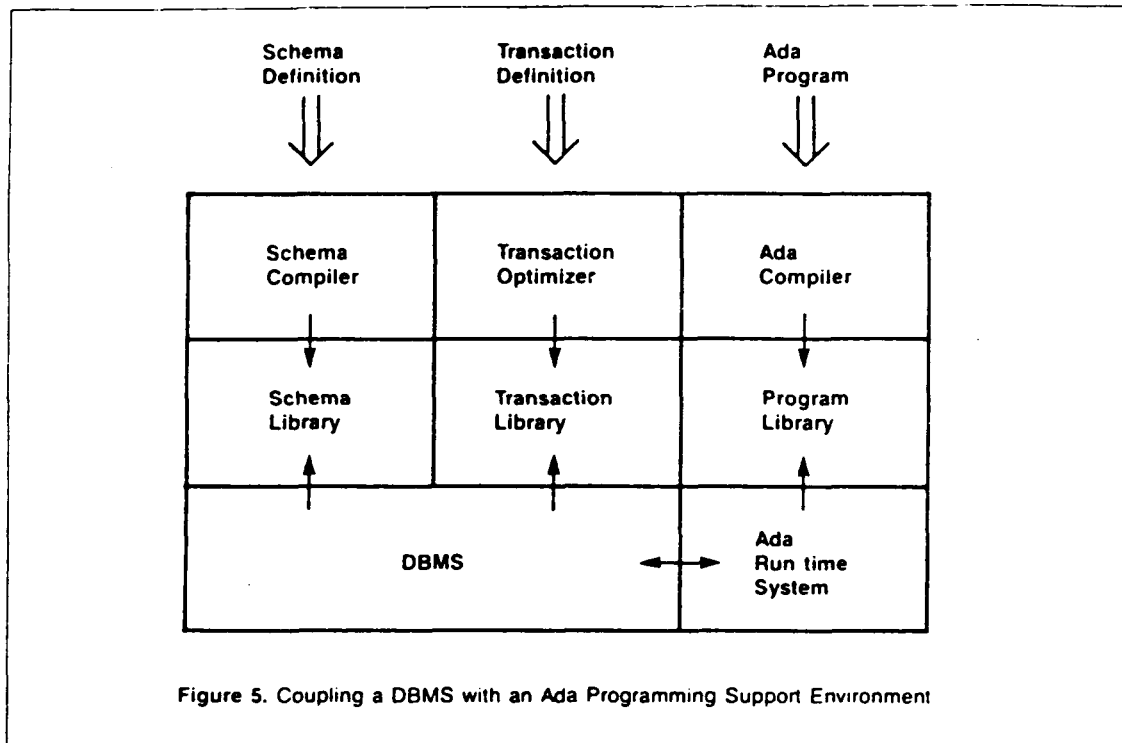
Figure 4. An Equivalent SQL Query

4. Adaplex

Database environments for popular programming languages, notably C, PL/1, COBOL, and Pascal, have resulted in extensions to the host programming language. At the outset, it was not clear whether Ada would also need to be extended to accommodate database applications. This is because Ada contains important new features not found in previous widely-used languages. In particular, Ada's package construct offers the potential for defining a database extension within the language itself.

There have actually been a number of proposals for coupling database management capabilities to Ada through the package construct [HTVN81, NOKI83, VINE83]. However, we feel that such approaches sacrifice usability and data integrity for not extending Ada [SCDF85]. Since our goal is to design the best Ada compatible language environment for developing database application programs, it is our desire to express as much of the database environment in Ada as possible, although not at the expense of database capabilities and ease of use.

Two major capabilities that must be provided by a database programming environment are schema definition (for describing the contents of the database) and transaction definition (for specifying operations on the stored data). In order to support database applications programming in Ada, it is necessary to couple the DBMS to an Ada programming support environment. One possible approach for achieving such a coupling is illustrated in Figure 5. Notice that both schema definition and transaction definition are separated from the Ada

Schema
Definition

Transaction
Definition

Ada
Program

Schema Compiler	Transaction Optimizer	Ada Compiler
Schema Library	Transaction Library	Program Library
DBMS		Ada Run time System

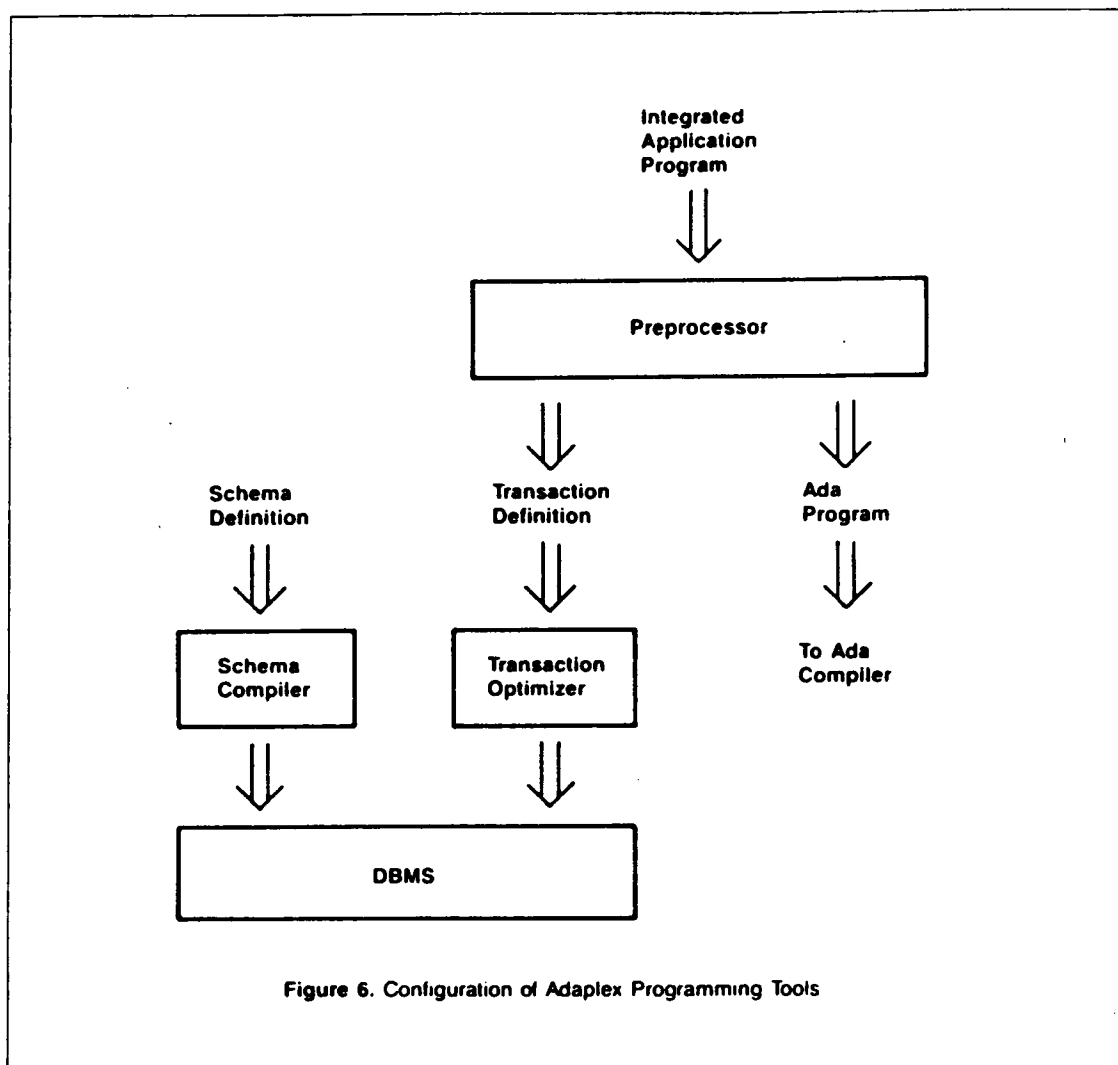Figure 5. Coupling a DBMS with an Ada Programming Support Environment

application program.

This separation works for database schema definition since the output of
the schema compiler can be logically thought of as an Ada package containing
type definitions representing a database schema. The separation of transaction
definition from application program is less natural because parameters must be
passed from the application program to the DBMS and transaction results must
be bound to application program variables.

In the course of our project, two approaches for handling transaction
definition have been considered. The first approach is similiar to the one used
for schema definition. A transaction definition is passed to the transaction
optimizer which generates an Ada package that implements (i.e. calls the
DBMS to execute) the transaction. The package is then loaded with the appli-
cation program. This approach, however, leaves the applications programmer
with a rather complicated interface. The programmer must learn a transaction
definition language which is quite distinct from Ada. Besides, parameter pass-
ing between the application program and the package that implements the
transaction is cumbersome. Since Ada is a strongly typed language, it might be
necessary to use an intermediate representation like character strings for pass-
ing certain parameters. This has a number of drawbacks. First, the program-
mer must explicitly encode and decode these strings. Second, compile time
type checking cannot be performed on the contents of these strings. In gen-
eral, such a parameter passing mechanism can be quite inefficient.

These difficulties lead us to adopt a second approach which permits the
application programmer to embed transaction definitions directly in an Ada
program. The result is an integrated language, called Adaplex, which provides
a tight coupling between Ada and our transaction definition language. No
changes were made to existing Ada constructs. The new constructs that were
added are treated in an Ada compatible manner. The coupling is achieved at
the expression level. Applications programmers are free to use Ada

```
                        Integrated
                        Application
                        Program

                            ⇓

                   ┌─────────────────────────────┐
                   │         Preprocessor         │
                   └─────────────────────────────┘

                   ⇓              ⇓              ⇓

     Schema              Transaction        Ada
     Definition          Definition         Program

        ⇓                    ⇓                 ⇓

   ┌──────────┐        ┌──────────────┐    To Ada
   │  Schema  │        │ Transaction  │    Compiler
   │ Compiler │        │  Optimizer   │
   └──────────┘        └──────────────┘

        ⇓                    ⇓

   ┌─────────────────────────────────┐
   │              DBMS                │
   └─────────────────────────────────┘
```

Figure 6. Configuration of Adaplex Programming Tools

expressions, control structures, and subprogram calls within a transaction definition. Because of Adaplex's uniform syntax and semantics, we expect it to be very easy to learn and use by trained Ada programmers.

For portability reasons, a preprocessor is used to decompose applications programs written in Adaplex into a transaction part and an Ada program part. The transaction part is forwarded to the transaction optimizer and the Ada part to the Ada compiler. The preprocessor is a very powerful tool. It provides the same integrity checking across the application program/DBMS interface that the Ada compiler provides for an Ada program.

The schema compiler, transaction optimizer, preprocessor, and DBMS form the minimum set of program development tools required for the database environment. Their combined configuration is shown in Figure 6. Any one of the Multibase, LDM, DDM systems can be substituted in place of the box labelled DBMS. Provided all these tools are written in Ada, database schemas, application programs, and databases may be ported between Ada installations.

Fundamentally, Adaplex adds two constructs to Ada, the *database declaration* and the *atomic statement*. These constructs provide for schema definition and transaction definition respectively. A database declaration specifies the data objects in a database, the types of those data objects, and their

```
database UNIVERSITY is

    type DEPT_NAME is (CS, EE, MA);
    type YEARS is new INTEGER range 0 .. 120;
    UNKNOWN_AGE : constant YEARS := 0;

    type COURSE is
      entity
        TITLE      : STRING (1 .. 6)
        ROOM       : STRING (1 .. 5);
        CREDITS    : INTEGER range 1 .. 4;
      end entity

    type PERSON is
      entity
        NAME       : STRING (1 .. 30);
        AGE        : YEARS := UNKNOWN_AGE;
        SSN        : INTEGER;
      end entity;

    subtype INSTRUCTOR is PERSON
      entity
        DEPT            : DEPT_NAME;
        COURSES_TAUGHT : set of COURSE;
      end entity;

    subtype STUDENT is PERSON
      entity
        DORM         : STRING (1 .. 10);
        ADVISOR      : INSTRUCTOR withnull;
        ENROLLMENTS : set of COURSE;
      end entity;

    overlap INSTRUCTOR with STUDENT;

    unique TITLE within COURSE;

end UNIVERSITY;
```

Figure 7. An Adaplex Database Declaration

consistency/integrity requirements. Database declarations are processed by the schema compiler. Figure 7 shows the database declaration for the university database that was depicted graphically in Figure 1. In addition to the type and subtype declarations, several constraint statements have been specified.

    overlap INSTRUCTOR with STUDENT;

indicates that it is legal for a PERSON entity to be both a STUDENT and INSTRUCTOR simultaneously.

    unique TITLE within COURSE;

indicates that all COURSE entities must have unique TITLEs.

```
with UNIVERSITY; use UNIVERSITY;
      .
      .
      .
ADD_COURSE:
  declare
    NEW_COURSE : COURSE;
  atomic
    NEW_COURSE := new COURSE (TITLE     = > "CS-101".
                             ROOM      = > GET_ROOM(CS).
                             CREDITS   = > 3);
    Include NEW_COURSE into
      COURSES_TAUGHT
        (I in INSTRUCTOR where NAME (I) = "Adam Jones");
  exception
    when UNIQUENESS_CONSTRAINT = >
      PUT_LINE("Duplicate course name");
  end atomic;
```

Figure 8. An Adaplex Database Transaction

A database is similar to a package since it is a related collection of data and type declarations. However, a database differs from a package in three principal ways. First, there are explicit protocols within Adaplex for several independent main programs to share the use of a database. Second, a strong discipline is imposed on the specifications allowed in a database declaration. Third, database declarations are developed interactively via the schema compiler, and they are stored for future reference in the schema library.

An atomic statement specifies a compound operation which must be indivisibly executed with respect to a database. The preprocessor extracts transactions from atomic statements for processing by the transaction optimizer. Figure 8 shows an Ada code fragment containing an atomic statement. This transaction creates a new COURSE entity and indicates that the course will be taught by the instructor named Adam Jones. Notice that the database type declarations are made visible by the with and use statements. The expression level integration of Daplex and Ada is illustrated by calling an Ada subprogram, GET_ROOM, to generate a value to assign to the ROOM function. Since COURSEs are constrained to have unique TITLEs, it is possible that the create statement may fail. An exception handler is included to cleanly handle this error.

An atomic statement is similar to a block in the sense that it is a compound statement that has associated declarations and exception handlers. However, an atomic statement differs from a block in three ways. First, atomic statements are executed indivisibly with respect to databases. Second, strong disciplines are imposed on the contents, nesting, parallel execution, and exception handling of atomic statements. Third, atomic statements are transformed by the preprocessor to extract database transactions.

A complete description of the Adaplex language can be found in [SFL83]. A detailed discussion on our rationale for developing Adaplex can be found in [SFL83, SCDF85].

5. LDM

LDM is a general purpose system for defining, storing, retrieving, updating, sharing, and protecting formatted information. While its users may be geographically distributed, LDM and its data must be centrally located. LDM is designed to provide all the functions typically found in a modern database system, including:

- logical and physical database definition,
- logical and physical database reorganization,
- a fully integrated data dictionary facility,
- an authorization mechanism for controlling database access,
- optimized selection of access paths for transactions,
- interference-free concurrent access by multiple users/transactions,
- automatic recovery from transaction failures, software crashes, and media failures,
- a dumping utility for taking a consistent snapshot of the entire database,
- a reload utility for restoring a database to a previously saved state.

LDM's main design objectives are transportability and high performance. Transportability is achieved by the use of Ada as the implementation language and by using a modular system architecture which is greatly facilitated by Ada's packaging construct and separate compilation mechanism. A description of LDM's component architecture can be found in [CFLR81]. High performance, on the other hand, requires the introduction of a number of technical innovations in the areas of physical data structuring, query optimization, concurrency control, and recovery management as identified below.

LDM is designed to provide complete physical data independence. It supports flexible physical structuring options so that a database administrator can tailor the physical representation of a database according to application requirements [CDFL82]. LDM employs special data structures for the efficient maintenance of referential integrity and other contraints associated with type overlaps in a generalization hierarchy. It also provides a wide range of options for the clustering of entities that belong to a generalization hierarchy. LDM supports dynamic data structures (namely, linear hashing [LARS80] and B-trees [COME79]) to eliminate the need for periodic reorganization. In order to support the efficient traversal of interentity references, LDM implements a pointer validation scheme that minimizes the updating costs associated with the use of dynamic data structures.

The design of LDM is geared towards the processing of repetitive transactions in a database applications programming environment. Transactions are compiled, thereby permitting the costs for parsing, authorization checking, and access path optimization to be amortized over multiple execution. LDM is also designed to optimize a much larger class of queries than relational systems. In particular, we have developed efficient strategies for processing queries with outerjoins and nested quantifiers [RCDF82, DAYA83A]. At the same time, the amount of effort that LDM will expend to optimize a transaction template can be controlled by a user (in the form of a pragma). Thus, a user can ensure that the effort for optimizing a given transaction template is commensurate with

the savings that can be expected to accrue over repeated execution.

LDM implements an integrated concurrency control and recovery mechanism which has the advantage of improving concurrency while simplifying transaction and system recovery. Specifically, LDM implements a multiversion mechanism that allows each read-only transaction to see a consistent snapshot of the database without having to synchronize with update transactions [CFLN82]. The essence of this mechanism is that update transactions create new versions of data objects without overwriting their previous versions. An efficient scheme is used to determine the appropriate version of different data objects each read-only transaction should see, and to identify those old versions that can be garbage collected. Since database dumps can be considered as read-only transactions that access the entire database, they can also be taken non-intrusively (i.e., without requiring the quiescence of concurrent updates).

In addition to being a stand-alone centralized database system, LDM also functions as an integral part of DDM and Multibase.

6. DDM

DDM is a homogeneous distributed database system built on top of a collection of LDMs running at different sites connected by a computer network. From the end-users' point of view, DDM performs precisely the same operations supported by LDM. This is because all complexities introduced by fragmentation, distribution, and replication of a database are hidden from end-users. Users access a distributed and replicated database in DDM just as they would access a centralized database in LDM. In a distributed environment, a copy of LDM and a copy of DDM are installed on each of several computers in a computer network where data is distributed / replicated. Each LDM is responsible for managing all locally stored data at its resident site. Each DDM cooperates with all other DDMs in the network in order to hide the distribution and replication of data from end users and applications. As a truly distributed system, DDM delivers the benefits of improved processing capacity, communications efficiency, survivability, and modular upward scaling. DDM provides the following important facilities.

- An integrated global schema that encompasses data stored at all sites. DDM maintains a global directory in order to keep track of the distribution and replication of data. It automatically maps transactions on the global schema into subtransactions on data stored at individual LDMs.

- Complete physical data independence. The database administrator is free to tune parameters involving the physical distribution, replication, and representation of the stored data, without affecting the external view of the database.

- Mutual consistency of replicated data. Users deal with logical data only. Propagation of updates to redundant copies of updated data is managed by the system.

- Atomicity of distributed transactions. DDM guarantees than no partial effects of one transaction will be seen by another. If a transaction is unable to complete, all of its effects on the database are automatically undone.

- Continued operation in spite of site failures. Users can continue to perform retrieval and update operations, even though some copies may be temporarily inaccessible. These latter copies are brought up to date by the system before being used for processing subsequent transactions.

- Dynamic integration of new sites. No quiescence of on-going activities is needed for reconfiguration of the system.

As in LDM, our main design objectives for DDM are transportability and performance. Again, we have introduced a number of technical innovations in the areas of data allocation, query optimization, concurrency control, and recovery management in order to obtain good performance. These are summarized below.

DDM supports flexible database fragmentation and allocation that can be used to improve locality of reference and efficiency of query processing [CDFR83]. Each database managed by DDM is optionally divided into a number of groups of data fragments, based on the likelihood of their being used together. Each group of data fragments constitutes a unit for allocation and may optionally be replicated at as many sites as desired. For a replicated fragment group, two kinds of copies are distinguished. Online copies are used for processing transactions. Offline copies serve as warm standbys that can quickly (and automatically) be upgraded to online status in order to retain a desired degree of resiliency as sites storing online copies fail. When specifying the replication parameters for a fragment group, a database administrator indicates the number of desired online copies and those sites whose copies are to be kept online preferrably. DDM will then strive to keep those copies at the preferred sites online, but dynamically bringing copies stored at other sites online to maintain the desired level of resiliency when necessary.

Unlike previous systems, DDM is designed to take into consideration database fragmentation and replication in its selection of strategies for processing transactions [CDFG83]. Whereas most previous studies on distributed query optimization assume the distribution of joins over unions, DDM will consider the options of using left distribution, right distribution, or no distribution at all when processing queries that involve such operations. DDM treats each fragment group as an integral data unit during the optimization process. Both compile time and run time optimization are performed. Compile time optimization seeks to identify a good order for processing the high level data manipulation operations on fragment groups without binding operations and copies to sites. This is because the choice of which copy of a fragment group to use for processing a transaction cannot be made until the availability of sites at run time is known. By dividing the optimization into two stages, DDM maximizes the amount of preanalysis done at compile time while ensuring the validity and optimality of the generated access plans.

DDM's concurrency control mechanisms are extensions of those used in LDM. Again, a multi-version mechanism is used to eliminate conflicts between read-only and update transactions [CG85]. In addition to improving parallelism, this mechanism greatly facilitates the taking of global checkpoints. Such a checkpoint may be necessary if one wants to reset a distributed database to a previous globally consistent state after the log data in one or more sites is damaged. With respect to replica control, DDM provides a balance between synchronization overhead and failure resiliency. Essentially, updates are propagated to online copies synchronously. Offline copies are only updated in a background batched fashion.

Because DDM is designed for distributed command and control applications, survivability is a very important issue. A special transaction commit algorithm is used to ensure that distributed transactions are terminated in a timely fashion, even in the presence of site failures, so that resources at the remaining operational sites can be fully utilized (without being tied down by incomplete transactions). DDM is designed to recover automatically from total failures wherein all of the sites coordinating a transaction or all of the sites storing replicated copies of a fragment group fail simultaneously. Previous systems have treated such failures as catastrophes and required human intervention for recovery. In order to speed up the availability of data at a recovering site, DDM employs an incremental site recovery strategy. Essentially, the fragment groups stored at the recovering site are prioritized and brought up to date one at a time (with the assistance of other replication sites). As soon as a fragment group is brought online, it can be used for processing new transactions without having to wait for the recovery of other fragment groups.

7. Multibase

Multibase is designed to provide a logically integrated, retrieval-only, user interface to a physically nonintegrated environment containing pre-existing databases. These databases may reside on different types of database management systems, at different physical locations, and on different types of hardware.

Before local databases can be accessed through Multibase, the local host systems must be connected to a communications network. This network can be local or geographically distributed. After Multibase has been connected to the same communications network, a global user can access data in the local databases through Multibase using a single query language. Each local site maintains autonomy for local database updates. Local applications can continue to operate using the existing local interfaces, as before.

Multibase presents the end user or application program with the illusion of a single, integrated, non-distributed database. Specifically, Multibase assumes the following responsibilities:

- providing a global and consistent picture of the available data,

- knowing the locations for the database items,

- transforming a query expressed in the global query language into a set of subqueries expressed in the different languages supported by the target systems,

- formulating an efficient plan for executing a sequence of subqueries and data movement steps,

- implementing an efficient plan for accessing the data at a single target site,

- moving the results of the subqueries among the sites,

- resolving incompatibilities between the databases (such as difference in naming conventions and data types),

- resolving inconsistencies in copies of the same information that are stored in different databases, and

- combining the retrieved data to correctly answer the original request.

Multibase has three key design objectives: generality, compatibility, and extensibility. To satisfy the first objective, Multibase has been designed to be a general tool, capable of providing integrated access to various database systems used for different applications. Multibase has not been engineered to be an interface for a specific application area. The second requirement of Multibase is that it co-exists and be compatible with existing database systems and applications. No changes or modifications to local databases, DBMS's, or application programs are necessary to interface Multibase with systems already in operation. The local sites retain full autonomy for maintaining the databases. All local access and application programs can continue to operate without change under Multibase. The third design objective is that it must be relatively easy to couple a new local system into an existing Multibase configuration.

All these objectives are achieved by designing a modular architecture for Multibase and by making the system largely "description driven" [LR82]. Multibase's modular architecture isolates those parts of the system that deal with specific aspects of a local system. Because of this, a Multibase configuration can be expanded to include a new DBMS in a short period of time and with little impact on the existing Multibase software. Descriptions are used throughout Multibase to tailor general modules for specific applications, users, and databases. These descriptions are written by the database administrator(s) who is responsible for tailoring a Multibase configuration.

Figure 9. Multibase Component Architecture
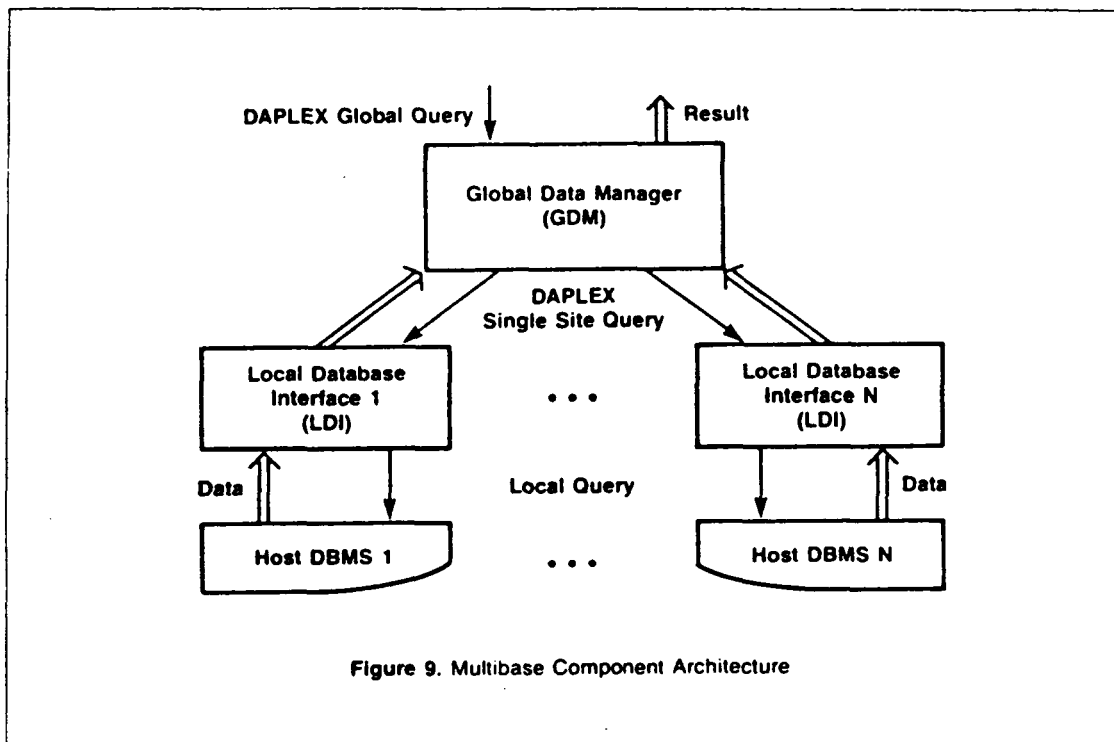
The component architecture of Multibase is illustrated in Figure 9. There are two types of modules: a global data manager (GDM) and a local database interface (LDI). All global aspects of a query are handled by the GDM. All specific aspects of a local system are handled by an LDI. There is one LDI for

each local host DBMS accessed by Multibase. The GDM makes use of LDM as an internal DBMS to manage its workspace. The LDM is used to store the results of the Daplex single-site queries which are processed by the LDI's and to perform all the required steps of the final query for combining and formatting the data.

It should be mentioned that Multibase does not provide the capability to update data in the local databases or to synchronize read operations across several sites. This is because implementing global concurrency control mechanisms for read or update operations would have necessitated the global process to request and control specific resources offered by the local systems (i.e., locking local database items) as required to ensure consistency across the databases. However, most systems do not make available to an external process the services necessary to implement global concurrency control. Since Multibase is designed to operate without requiring modifications to existing systems, the tools necessary to ensure consistency across databases are not globally available. Thus, autonomy of database update is maintained locally, and Multibase provides the global user with the same level of data consistency that the local host DBMSs provide to each local database user.

In addition to the highly modular and description driven architecture, the design of Multibase has required research in the areas of schema integration, global query optimization, and local query optimization. Our results in each of these areas have been reported in [KG81, DAYA84a], [DAYA83b, GY84, DAYA84b], and [DG82] respectively.

8. Status

Designs of the Daplex and Adaplex languages are complete. Prototype versions of Multibase and LDM which support most of the described capabilities have been implemented. Implementation of DDM is well underway. To date, the systems contain approximately 500,000 lines of Ada source code. Most of the implementation was done in an Ada-subset using an Ada-to-Pascal translator [SOFT81]. The systems were then converted to full Ada using the DEC VAX Ada compiler [DEC85]. Development is continuing using both VAX Ada and Rational's Ada Development Environment [RAT85]. The initial target environment for all three systems is VAX VMS. The current systems support an interactive version of Adaplex (i.e., Daplex).

9. References

[CDFG83b]
A. Chan, U. Dayal, S. Fox, N. Goodman, D. Ries, D. Skeen. "Overview of an Ada Compatible Distributed Database Manager." *ACM SIGMOD Conference Proceedings*, 1983.

[CDFL82]
A. Chan, S. Fox, S. Danberg, W. Lin, A. Nori, D. Ries. "Storage and Access Structures to Support a Semantic Data Model." *VLDB Conference Proceedings*, 1982.

[CDFR83]
A. Chan, U. Dayal, S. Fox, D. Ries. "Supporting a Semantic Data Model in a Distributed Database System." *VLDB Conference Proceedings*, 1983.

[CFLN82]
A. Chan, S. Fox, W. Lin, A. Nori, D. Ries. "The Implementation of an Integrated Concurrency Control and Recovery Scheme." *ACM SIGMOD Conference Proceeding*, 1982.

[CFLR81]
A. Chan, S. Fox, W. Lin, D. Ries. "The Design of an Ada Compatible Local Database Manager." Technical Report CCA-81-09, Computer Corporation of America.

[CG85]
A. Chan, R. Gray. "Implementing Distributed Read-only Transactions." To appear in *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 1, February 1985.

[DATE81]
C. Date. "Referential Integrity." *VLDB Conference Proceedings*, 1981.

[DATE84]
C. Date. *A Guide to DB2*, Addison Wesley, 1984

[DAYA83a]
U. Dayal. "Processing Queries with Quantifiers: A Horticultural Approach." *ACM PODS Conference Proceedings*, 1983.

[DAYA83b]
U. Dayal. "Processing Queries over Generalization Hierarchies in a Multidatabase System." *VLDB Conference Proceedings*, 1983.

[DAYA84a]
U. Dayal, H. Hwang. "View Definition and Generalization for Database Integration in Multibase: A System for Heterogeneous Distributed Databases." *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 4, November 1984.

[DAYA84b]
U. Dayal. "Query Processing in a Multidatabase System." in *Query Processing in Database Systems*, (W. Kim, D. Batory, D. Reiner, editors), Springer Verlag, 1984.

[DEC85]
Digital Equipment Corporation. "Developing Ada Programs on VAX/VMS." 1985.

[DG82]
U. Dayal, N. Goodman. "Query Optimization for CODASYL Database Systems." *ACM SIGMOD Conference Proceedings*, 1982.

[GSCD83]
N. Goodmam, D. Skeen, A. Chan, U. Dayal, S. Fox, D. Ries. "A Recovery Algorithm for a Distributed Database Management System." *ACM PODS Conference Proceedings*, 1983.

[GY84]
D. Goldhirsh, L. Yedwab. "Processing Read-Only Queries over Views with Generalization." *VLDB Conference Proceedings*, 1984.

[HD84]
H. Hwang, U. Dayal. "Using Semiouterjoins to Process Queries in a Multidabase System." *ACM PODS Conference Proceedings*, 1984.

[HTVN81]
J. Holland, K. Tai, M. Van Name. "An Ada Relational Database Interface Using Abstract Data Types." TR 81-07, North Carolina State University, 1981.

[KG81]
R. Katz, N. Goodman. "View Processing in Multibase — A Heterogeneous Database System." in *Entity-Relationship Approach to Information Modelling and Analysis*, (P. Chen, editor), ER Institute, Saugus, CA, 1981.

[LR82]
T. Landers, R. Rosenberg. "An Overview of Multibase." in *Distributed Databases*, (H. Schneider, editor), North Holland Publishing Company, 1982.

[NOKI83]
Nokia Data Terminal Systems. "MPS 10 Database Management System Functional Description." Version 1.0, June 1983.

[RAT85]
Rational, Inc. "Rational Environment Reference Summary." 1985.

[RCDF82]
D. Ries, A. Chan, U. Dayal, S. Fox, W. Lin, L. Yedwab. "Decompilation and Optimization of Adaplex: A Procedural Database Language." Technical Report, Computer Corporation of America, 1982.

[SCDF85]
J. Smith, A. Chan, S. Danberg, S. Fox, A. Nori. "A Tool Kit for Database Programming in Ada." To appear in *International Ada Conference Proceedings*, 1985.

[SFL83]
J. Smith, S. Fox, T. Landers. "Adaplex: Rationale and Reference Manual." Technical Report, Computer Corporation of America, 1983.

[SLRR84]
S. Fox, T. Landers, D. Ries, R. Rosenberg. "Daplex User's Manual." Technical Report CCA-84-01, Computer Corporation of America, March 1984.

[SOFT81]
SofTech, Inc. "Interim Ada-to-Pascal Translation Tool Language Reference Manual." TP 124, September 1981.

[SWK76]
M. Stonebraker, E. Wong, P. Kreps. "The Design and Implementation of INGRES." *ACM Transactions on Database Systems*, Vol. 1, No. 3, September 1976.

[VINE83]
        D. Vines, Jr. "An Interface to an Existing DBMS from Ada (IDA)." GTE
Network Systems, 1983.