The C-based Database Programming Language Jasmine/C

M. AOSHIMA, Y. IZUMIDA * , A. MAKINOUCHI**, F. SUZUKI * , and Y. YAMANE
Fujitsu Laboratories Ltd. * Fujitsu Ltd.: 1015 Kamikodanaka, Nakahara-ku, Kawasaki 211, JAPAN ** Kyushu University, Department of Computer Science and Comm. Eng.: 6-10-1, Hakozaki, Higashi-ku, Fukuoka 812, JAPAN

ABSTRACT

Jasmine/C is a C-based database programming language that allows the handling of persistent objects in Jasmine databases. The language is used to write methods for objects and application programs. Both navigational and associative access to objects are supported. Dot notation is used for Jasmine/C queries. Attributes of different (but linked) objects are concatenated in SQL-like queries free of from-clauses. This allows the joining of objects without explicit join-predicates. Several new features are introduced into the architecture of Jasmine. Memory KB is a memory-based database where each object is accessed via a pointer. Tuples in XDE (the lower layer of Jasmine), when in database buffers, can also be accessed via pointers. NF^2 tables are supported for clustering values for multiple-valued attributes; these allow faster execution of Jasmine/C programs.

1. Introduction

Jasmine/C is a C-based database programming language that allows the handling of persistent objects in Jasmine databases. It was designed and developed so that users of Jasmine/C databases may define, retrieve, and manipulate persistent objects in the same way as they define and manipulate C structured data. It is used to write methods for objects as well as application programs. It is also used to write non-database application programs. For such applications, users may take advantage of the object-oriented programming style.

Jasmine is an object-oriented database system. The system was designed to meet three requirements: (1) to provide AI people with the means to manage a large amount of knowledge, (2) to provide DB people with the means to model complex objects [KIMW87] used in engineering environments, and (3) to define and manipulate persistent objects as easily as structured data in the C language.

The first requirement suggests the use of the frames [MINS75] for knowledge representation. The frame is a data structure to which procedures, called demons, can be attached for integrity maintenance and reasoning. Many knowledge representation languages have been frame-based, and some were actually implemented [STEF86]. One early example is Loops. In this language, different programming styles such as rule-based programming and access-oriented programming are integrated using the object-oriented paradigm. Other current AI shell languages follow this trend so that static data structures, procedures, and rules can be mixed to model domain knowledge [STEF84].

A drawback of such languages is that they do not provide appropriate facilities for managing a large amount of shared knowledge. The factual knowledge stored in databases may be accessed via the usual database commands, but it is not integrated in the knowledge representation framework, and thus cannot be managed uniformly.

The success of the relational data model is due to the simplicity of the model, and easy-to-use associative query languages. In the model, relations perform the double duties of data specification and data container. Users may retrieve data by searching the containers. The search itself can be specified with conditions

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

which the data to be retrieved must satisfy. In designing Jasmine/C, we thought that these basic ideas should be inherited, for the reason that in shared large databases without condition-based search, users become easily lost before accessing the data they require. This led us to the integration of SQL-like but simplified 'where-clauses' into C. This necessitates set-oriented handling of objects - a designe problem, since C is not a set-oriented language.

In programs, complex objects are usually represented by structured data types, where a structure may be defined in terms of other structures. This facility is augmented by pointers or references without which recursive data types can not be implemented. Similarly, references to persistent objects are supported by Jasmine/C. Objects are identified and referenced using object identifiers (OIDs). OIDs play the same role as the pointer values (addresses) by which program data are referenced. This allows users to describe persistent complex objects and to navigate from object to object via OID linkages. This correspondence between C structured data and Jasmine/C persistent objects satisfies the above-mentioned requirement (3), and helps bridge the gap between programming in C and programming in Jasmine/C.

However, there are three differences between the C structured data and the Jasmine/C persistent objects. The first is that functions in the sense of the C language can be constituent members of persistent objects while in the C strucure they can not. The second is that associative retrieval of persistent objects is provided in Jasmine/C as a basic facility. The third is that each member of a persistent object can be annotated as are slots in a frame. This gives Jasmine/C a modeling capability as powerful as that of frame-based AI languages.

In this paper, examples are used to clarify the discussion. They are borrowed from [ATKI87] with slight modification, and represent the functionality that any database programming language is expected to support.

2. Classes and Types

Classes in Jasmine are collections of similar objects. Jasmine distinguishes two kinds of object: one is immediate and the other is referential. Numbers and character strings are immediate. Immediate objects do not have separate OIDs, but 'represent themselves'. They can be copied to any location in the database. In contrast, referential objects have separate OIDs. OIDs are not changed during the lifetime of their objects. A referential object can not be copied without changing its OID. Users of Jasmine/C may use OIDs to refer to objects that have them.

Attributes of an object are defined as functions in the sense of the functional data model [SHIP81] . Functions in Jasmine are defined in terms of two classes, domain and range. Objects of the domain class are arguments of functions, and objects of the range class are the values returned by functions. Fig. 1 illustrates a sample database. Attribute Sno is defined as a function whose domain and range are classes SUPPLIER and INTEGER, respectively. Attribute Supply's domain class is SUPPLIER and its range class is BASEPART. In principle, functions in Jasmine have only one argument. This is different from the functional data model, where Cartesian products of sets can be the domain.

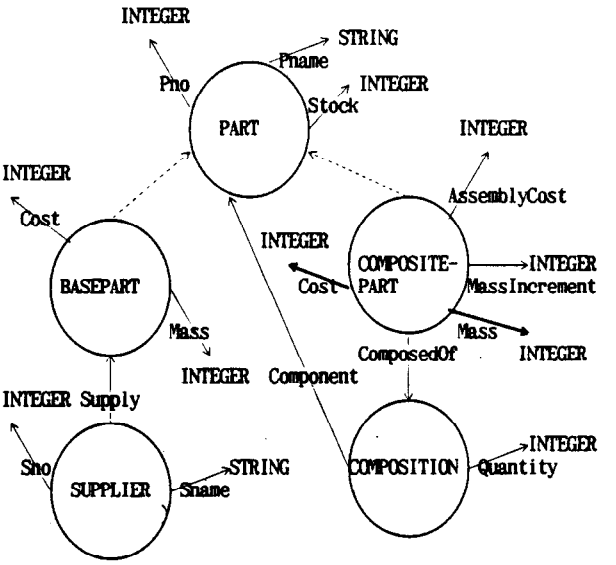

Fig.1 PARTS database. Classes of referential objects are shown with circles. Named arrows denote attributes. Bold arrows denote procedural attributes. Superclasses are indicated by dotted arrows.

In Jasmine/C programs, a dot '.' is used to denote 'an application of attribute (i.e., function)' to an object or a collection of objects in the domain class. We use the term 'set' for a collection, although it allows the

duplication of members. For example, if 's' represents a supplier, 's.Supply' means 'Supply(s)'. We call an object such as 's' a 'receiver', and say that message 'Supply' is sent to it [GOLD83].

```
void CreatePartsDb() {
  KB partskb;
  TRANS mytrans;
  CLASS partcls, bpartcls, cpartcls, suppliercls;
  partskb=KB.Open(PARTS);
  mytrans=TRANS.Begin();
  partcls=CLASS.New(
    kb partskb
    name PART
    super COMPOSITE
    property INTEGER Pno mandatory
             constraint {/* See Fig.6 for the def.*/ }
             STRING Pname
             INTEGER Stock
              if_updated {/* See Fig.7 for the def.*/ } );
  bpartcls=CLASS.New(
    kb partskb
    name BASEPART
    super partcls
    property INTEGER Cost
             INTEGER Mass
    instance_method VOID Delete inherited
                after demon uninheritable
                {/* See Fig.8 for the def.*/ } );
  cpartcls=CLASS.New(
    kb partkb
    name COMPOSITEPART
    super partcls
    property INTEGER AssemblyCost
             INTEGER MassIncrement
             COMPOSITION ComposedOf multiple
    instance_method INTEGER Cost()
                      {/* See Fig.3 for the def.*/ }
                    INTEGER Mass()
                      {/* The definition is ommitted.*/} );
  suppliercls=CLASS.New(
    kb partkb
    name SUPPLIER
    super COMPOSITE
    property INTEGER Sno mandatory
             BASEPART Supply multiple
             STRING Sname);
      .
      . /* Code for class COMPOSITION is omitted */
      .
  mytrans.End();
  partskb.Close();
}
```

Fig.2 A Jasmine/C program to create classes of the sample database shown in Fig.1. For simplicity, only part of the code is shown.

The value that an attribute returns is either an object or a set of objects. The returned value is called an attribute value. An attribute is either single-valued or multiple-valued. A multiple-valued attribute may return the empty set. The empty set means 'nothing' and is denoted in Jasmine/C as ' {} '. Although we define a class as a set (i.e., extension) of objects, the set of objects making up an attribute value is not a class.

```
INTEGER Cost() {
  INTGER cost;
  COMPOSITION cmpts multiple, cmpt;
  SCAN scan;
  PART part;
  cost=0;
  cmpts=self.ComposedOf;
  scan=cmpts.Openscan();
  while(!(scan==VOID))  {
    cmpt=scan.Next();
    part=cmpt.Component;
    if (part.Class == <BasePart>)
      cost+=part.Cost*cmpt.Quantity;
      else
      cost+=part.Cost()*cmpt.Quantity;
  }
  scan.Close();
  cost+=self.AssemblyCost;
  return(cost);
}
```

Fig.3 The definition of instance_method Cost, a recursive function.

In Jasmine, there are two ways to define attributes. One way is to enumerate all possible pairs of receiver and attribute value. An attribute defined in this way is called an attribute by enumeration or a 'property'. Properties can not be defined on classes of immediate objects such as INTEGER. That is, such classes can not be domains for property type functions. A reverse function can be defined from a property if the property is defined as a function between two classes of referential objects. Its domain and range classes are the range and domain classes of the property, respectively. Since a reverse function is a function, it is, by definition, an attribute of the domain class of the reverse function. For example, the reverse attribute of Supply, notated '-Supply', is defined on class BASEPART. Definition by enumeration is the traditional way to define values in databases.

The other way to define attributes is to write a C-style procedure or function. The value of the attribute is obtained by executing the procedure. Such an attribute is called a procedural attribute. Procedural attributes are equivalent to methods in object-oriented programming languages. The methods are written in Jasmine/C, an extension of the C language. The attribute 'Cost' defined on COMPOSITEPART, shown in Fig.2, is an example. The procedure body is defined in Fig.3. This makes Jasmine 'computationally complete' [ATKI89], whereas the functional data model [SHIP81] is not, for it presupposes a restricted query language with which derived functions are defined.

We have two kinds of attributes, property and procedural attributes, which are defined in different ways. We can also classify attributes according to the mapping between receivers and attribute values: (1) one-to-one, (2) one-to-many, (3) many-to-one, and (4) many-to-many. To discuss this more formally, let us assume that d_i (i=1, 2,..., n) and r_j (j=1, 2,...,m) are objects in the domain and range classes of attribute f, respectively. We will use ' { } ' to denote a set. We may formalize the four classifications as follows: (1) d_i .f = r_j (one-to-one), (2) d_i .f = $\{r_1, r_2,..., r_m\}$ (one-to-many), (3) $\{d_1, d_2,..., d_n\}$ .f = r_j (many-to-one), and (4) $\{d_1, d_2,..., d_n\}$ .f = $\{r_1, r_2,..., r_m\}$ (many-to-many)

Procedural attributes may be of any of the four classifications, but properties must be one-to-one or one-to many. However, an attribute, either property or procedural, can be applied to a set of objects in a way different from the many-to-one and many-to-many mappings explained above. This makes it possible for properties to simulate many-to-many mappings in a restricted way: (5) $\{d_1, d_2,..., d_n\}$ .f = $\{r_1, r_2,..., r_n\}$, where for each i, $r_i = d_i$ .f. These semantics enable the simulation of (and therefore the elimination of) loops of the following form: 'for (i=1; i<=n;;) {... r_i =d_i .f; ... } '. Note that in (4), r_i is not necessarily equal to d_i .f.

The difference in implementation between (4) and (5) is stated as follows. In the case of (4) (and (3)), the procedural attribute f is called once. When it is called, the procedure takes the receiver, a set of objects, as its argument and processes it. In the case of (5), attribute f is called for each receiver d_i to return value r_i. From this observation, (4) is expected to show better performance than (5) due to the overhead in calling procedures.

To realize (3) and (4), Jasmine/C distinguishes instance-methods and set-methods for procedural attributes. In instance-methods, variable 'self' receives an object representing the receiver, while in set-methods, it receives a set of objects. For example, the system-defined variable 'self' in instance-methods Mass and Cost, defined on class COMPOSITEPART in Fig.2, is supposed to receive an object of class COMPOSITEPART. This does not mean, however, that they cannot be applied to a set of objects, as mentioned above. In contrast, a set-method is also applicable to an object. An object is interpreted as a set with one member.

The set-method of type (3) is needed to implement aggregate functions such as Sum, Max, Count, and Average. They are all applied to a set of numbers and return a number. The sort function is of type (4). It takes a set of objects and returns a sorted set of objects.

As a third method, the class-method is introduced in contrast with instance- and set-methods. A class-method is defined on a class. When the class is designated at the place of the receiver of a message invoking the method, the receiver is not objects of the class, but the class itself. This is necessary to write methods such as 'New'. 'CLASS.New()' in Fig.2 and 'COMPOSITEPART.New()' in Fig.5 are examples in which instance objects are created for the classes CLASS and COMPOSITEPART. Note that if these 'New's are defined as being either a instance-method or a set-method, they are

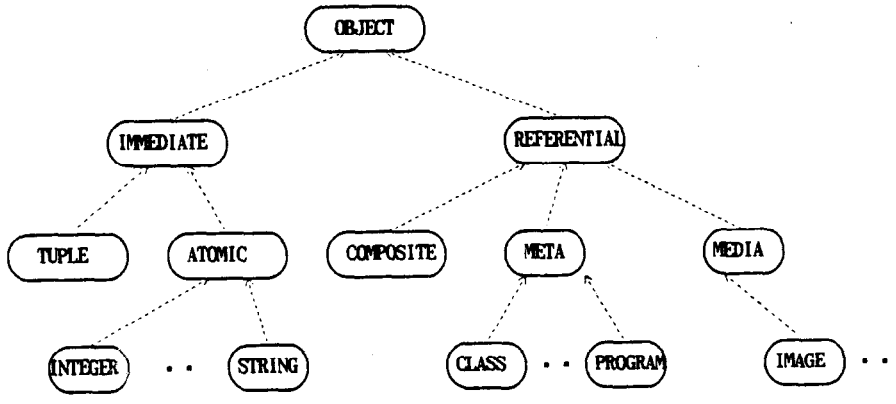

Fig.4 A part of the Jasmine class hierarchy.

applied to all instance objects of CLASS or COMPOSITEPART following (3), (4), or (5) semantics.

A class can be divided into subclasses. In relation to its subclasses, the divided class is called a 'superclass'. A subclass inherits all attributes of its superclass. A subclass can have attributes which its superclass does not have. Subclasses can be further divided, creating a class hierarchy.

The only class hierarchy that Jasmine allows is a tree structure, and thus, multi-inheritance is not permitted. In this way, we avoid needless semantical and implementation complexity - we have not found any useful and practical application for which it was necessary.

The topmost class of the hierarchy is OBJECT, followed by IMMEDIATE and REFERENTIAL as shown in Fig.4. The distinction between the last two classes comes from whether their objects have OIDs or not, as explained already. Other differences are: (a) Properties can not be defined on IMMEDIATE and its subclasses (i.e., hereafter, the term 'subclasses' will refer not only to direct subclasses but also descendant subclasses). (b) Classes under REFERENTIAL are containers or clusters of objects which may be searched. Immediate object classes being but types are not.

Henceforth, we will focus on referential objects, and will use the terms 'object' and 'class' to mean 'referential objects' and 'referential class', unless stated otherwise.

The Jasmine class hierarchy is also a set-inclusion hierarchy. This means a class contains its own instance objects as well as the objects contained by its subclasses. In other words, the contained objects are said to belong to or to be members of the containing class. An object is born as an instance object of a class when the message 'New()' is sent to the class.

The type of a class is defined to be the set of the attributes (i.e., functions) applicable to the class. The type of an object is the type of the class of which it is an instance.

Any attribute can be annotated, in the same manner as slotes in a frame are annotated by facets [MINS75]. Facets such as 'if-needed', 'if-updated', and so on that attach 'demon' procedures to an attribute are very useful for maintaining integrity of databases. In Jasmine, users can control the inheritance of each facet separately from the attribute inheritance. These topics are discussed in Section 4.

3. Navigation and Associative Retrieval

One of the characteristics of object-oriented database systems is that objects can be directly referred to by their identifiers. In Jasmine/C, object identifiers (OIDs) are returned when creating new objects or retreiving objects from the databse. They are assigned to object variables just as values in C programs are assigned to variables.

Declaration of object variables and navigation

An object variable is declared with a class. When the keyword 'multiple' follows the declaration, a set of objects can be assigned to the variable; the variable is then said to be multiple-valued. Otherwise, the variable is called single-valued. Let us suppose x to be an object variable declared as 'C x;', where C is a class. The declaration states that x represents some object belonging to class C. If x is declared as 'C x multiple;', any member of the assigned value of x belongs to class C. As stated in the former section, objects belonging to a subclass of C belongs to C, too. So, in Jasmine/C, the type of an object variable declared using a certain class is the union of the types of the class and its subclasses. For example, the declaration 'PART x;', where PART is a class shown in Fig.1, declares variable x as one to which either part, basepart, or composite part objects (its OIDs) may be assigned.

First, let us create the sample database of Fig.1. The program for this is shown in Fig.2. In Jasmine, the logical storage space unit is called KB, an abbreviation of Knowledge Base. (Recall that the space is used to store frames.) A KB is a segment within which classes are created. Any number of KB can be used. A class cannot be extended over more than one KB, but a tree of classes can be. A KB is created and managed by class KB. 'KB.Open(PARTS)' opens the already-created KB named PARTS, and returns its OID. The OID is assigned to variable partskb. The variable can be used in other statements to identify the object PARTS.

In Jasmine, everything is an object. For example, a transaction is an object of TRANS, a system-defined class. 'TRANS.begin()' starts a transaction and returns its OID. The OID is used by 'mytrans.End()' to close the transaction.

Classes PART, BASEPART, and COMPOSITEPART are created by sending messages to class CLASS. CLASS is a system-defined class that holds meta-information concerning classes, including itself. 'New' is a procedural attribute defined on CLASS as a class-method. The attribute returns the OID of the created class. PART is the superclass of both BASEPART and COMPOSITEPART. Variable partcls is passed as a keyword parameter in message 'New' to denote this fact. Sometimes, it is preferable to use names instead of variables. For example, users can write 'partcls = CLASS.New(...super PART...) ;' to indicate that the superclass is PART. This description leads to runtime overhead in searching class CLASS for the PART object. Taking full advantage of direct reference while preserving reference by value is one design strategy for Jasmine/C.

```
void NewComposite()  {
  KB partskb;
  TRANS mytrans;
  INTEGER partnum, quant, asscost, incrmass;
  STRING partname;
  PART newpart, cmpntpart;
  COMPOSITION newcomp;
   ┌─ Interaction to users (1) ──────────────────────┐
   │ Pno, Pname, AssemblyCost, and MassIncrement are │
   │ taken as input and assigned to partnum, pname,  │
   │ asscost, and incrmass.                          │
   └─────────────────────────────────────────────────┘
  partskb=KB.Open(PARTS);
  mytrans=TRANS.Begin();
  newpart=COMPOSITEPART.New(Pno partnum);
  newpart.AssemblyCost=asscost;
  newpart.IncrementalMass=incrmass;
  newpart.Pname=partname;
  partnum=0;  /* '0' can not be any part number. */
   ┌─ Interaction to users (2) ──────────────────────┐
   │ Pno and Quantity of each component part of the  │
   │ composite part in question are read into partnum│
   │ and quant.                                      │
   └─────────────────────────────────────────────────┘
  while ( partnum !== 0)  /* '0' means 'no more components' */
     {  cmpntpart=PART where PART.Pno == partnum;
        if (cmpntpart==VOID)  /* VOID means nothing. */
          { /* Code for error handling is omitted. */}
        newcomp=COMPOSITION.New();
        newcomp.Component=cmpntpart;
        newcomp.Quantity=quant;
        newpart.ComposedOf+=newcomp;/* Append newcomp */
          ┌─ Interaction to users (3) ──────────────┐
          │ Pno and Quantity of each component part │
          │ of the composite part in question are   │
          │ read into partnum and quant.            │
          └─────────────────────────────────────────┘
     }
   mytrans.End();
   partskb.Close();
}
```

Fig.5 An application program in Jasmine/C which creates objects for COMPOSITEPART and gives values to their attributes.

Instance objects are created by 'New' as shown in Fig.5. This 'New' is a procedural attribute defined on class REFERENTIAL and is inherited by all its subclasses. The more local 'New's of classes KB and CLASS override this more general one. In Jasmine, as described in Section 4, creating an object involves the insertion of a tuple into a table. The common functionality of the three 'New's is the insertion of such tuples. In addition, the 'New' of KB prepares storage space, and that of CLASS creates a table for the new class.

Fig.5 shows a Jasmine/C program that contains statements to create objects and to assign values to their attributes. Variable newpart is declared to be of class PART, and is assigned the OID of the newly created object that is returned by expression 'COMPOSITEPART.New(Pno partnum)'. The dot '.' in this expression and those in the assignment statements following the expression are used to designate an object attribute. This dot notation is chosen due to the similarity with the notation used to indicate a member of a C structure. With this notation, C programmers can manipulate persistent objects as if they were structured data. However, the semantics of the dot notation is extended to indicate the application of functions. This extension makes it possible to have procedures as constituent members of object structures. 'COMPOSITEPART.New (Pno partnum)' may be considered as a call to a C function named New, which is defined as a member of structure COMPOSITEPART. This makes Jasmine/C persistent objects different from C structures.

Multiple-valued attributes allow the introduction of new syntax for appending members to sets, or removing members from sets. The operators '+=' and '-=' are used for appending and removing, respectively. The statement 'newpart.ComposedOf += newcomp' in Fig.5 is an example.

Dots are also used to navigate from object to object. For example, let x be a variable, declared as 'SUPPLIER x;', and let a supplier object be assigned to it. Then 'x.Supply.Cost' denotes the cost of the basepart(s) which x supplies. In general, two attributes concatenated by a dot are said to form a composed attribute. The composed attribute is the composed function in the sense of the functional data model .

Associative retrieval

It is important for a database programming

language to have associative retrieval capability, since to know where the required data is to be retreived from and how to retrieve it in the large shared database is practically impossible without this capability. The success of the relational database model is mainly due to its associative retrieval languages. Our decision while designing Jasmine/C was to incorporate the concepts embodied in relational query languages such as SQL into the framework of C, using the object-oriented paradigm. These concepts are: (1) Classes are containers of objects, and are searched for objects satisfying a given predicate-based 'where-clause' conditions. Classes in Jasmine play the same role of relations in the relational model. (2) Multiple-valued variables are introduced, so that variables may be assigned to more than one object satisfying a given condition. Such a variable can be used as a container of objects from which objects are associatively retrieved. This is similar to the relational algebra, where the result of an operation, a relation, can be used as an operand for other operations. (3) Explicit join-predicates can be used in where-clauses. Note that dots sometimes represent implicit equi-join-predicates.

We shall now explain how these concepts are embodied in Jasmine/C, using examples. The following lines of Jasmine/C program retrieve all suppliers whose name is "Fuji", in the database of Fig.1:

```
SUPPLIER fujis multiple; /* Multiple-valued
                            variable fujis */
fujis = SUPPLIER where SUPPLIER.Sname
                       == "Fuji";
```

If fujis is declared as a single-valued variable, only one supplier from among the retrieved suppliers is assigned to it. In this example, the dot in 'SUPPLIER.Sname' is used to designate the SUPPLIER attribute Sname, whose value is checked against "Fuji". SUPPLIER indicates the search space where objects are checked against the predicate.

After this assignment, variable fujis can be used in two ways. First, it can be used to represent those objects having the name "Fuji". So, 'fujis.Sno' denotes the objects' Sno, and its value is a set of integers. This set can be assigned to a multiple-valued INTEGER variable, as in 'INTEGER snos multiple; snos = fujis.Sno; '. Similarly, the set of baseparts supplied by suppliers named "Fuji" is obtained from 'baseparts = fujis.Supply;', where 'baseparts' is a multiple-valued BASEPART variable.

Second, variable fujis can be used as a search space denoting a subset of the class with which the variable is declared. The following statement is to retrieve those basepart objects whose cost is more than $10, and are supplied by suppliers named "Fuji".

```
baseparts = fujis.Supply
            where fujis.Supply.Cost > 10;
```

This statement retrieves a set of objects (1) { x.Supply | x.Supply ∋ y.Oid and y.Cost > 10} , where x and y are variables over fuji, a subset of SUPPLIER, and BASEPART, respectively, and Oid is an object attribute whose value is OID. The symbol' ∋' , meaning set membership, is used, since Supply is a multiple-valued attribute having a set as its value. A similar statement, but with different meaning, can be written:

```
baseparts = BASEPART
            where fujis.Supply.Cost > 10;
```

This creates the set of basepart objects (2) { y | x.Supply∋ y.Oid and y.Cost > 10} . The difference between (1) and (2) becomes clear if a Fuji supplier supplies two baseparts whose cost are $3 and $11. (2) includes only the basepart whose cost is $11 while (1) includes both. The reason is that (1) is read as 'Get all baseparts supplied by each supplier x such that x supplies at least one basepart whose cost is over $10'. (2) means 'Get all baseparts whose cost is over $10 and for each of which there is some supplier.' This shows that the use of dot notation in the target part (i.e., the part that comes before the where-clause) extends the relational query language.

Explicit join predicates are permitted in where-clauses in Jasmine/C. The predicate 'fujis.Supply.Cost > 10' can be written as 'fujis.Supply == BASEPART.Oid and BASEPART.Cost > 10'. We use '==' (i.e., equality) instead of ' ∋ ', but the meaning is kept the same. Explicit join predicates are needed in such applications as the decision support system where asking ad hoc queries is common and essential. Consider the following example: 'SUPPLIER suppliers multiple; suppliers = SUPPLIER where SUPPLIER.Sname == SUPPLIER.Supply.Pname'. This query is to retrieve those suppliers supplying baseparts whose name is identical to that of the

supplier.

Procedural attributes can appear wherever properties can appear. This is a natural consequence of having adopted the functional data model as the basis of Jasmine data model. However, syntactically different notation must be used for procedural attributes, since we follow the C language syntax for calling functions. Parentheses are always needed. As will be explained in the next section, attributes Cost and Mass of COMPOSITEPART are procedural. They return the cost and mass of a composite part object, which are computed from the cost and mass of the object's components. The following query is to retrieve composite parts whose cost is less than $50: 'COMPOSITEPART compositeparts multiple; compositeparts = COMPOSITEPART where COMPOSITEPART.Cost() < 50;'. The syntax is inconvenient when it involves polymorphism [STEF84]. Let us suppose that parts, either basepart or composite, having cost less than $40 are to be listed. Since PART contains both BASEPART and COMPOSITEPART, the following query is valid if attribute Cost of BASEPART is procedural: 'PART parts multiple; parts = PART where PART.Cost() < 40;'. The procedural attribute to be added to class BASEPART is 'INTEGER Cost() { return self.Cost;} '.

Note that in the example in Fig. 1, Cost is not an attribute of class PART, although PART is expected to contain the objects having this attribute. A query of the form 'parts = PART where PART.Cost < 50;' is called incomplete in the sense that the query is incomplete in terms of attribute description. Incomplete queries are very convenient when the user's knowledge on the objects is incomplete or partial. Such situations often occur whenever we use databases made by others. Jasmine/C transforms incomplete queries into complete queries during compile time. Thus, the equivalent complete query 'parts = BASEPART where BASEPART.Cost < 50;' is substituted.

System programmers often use lists where structured data of different types are chained. They need to check the type of each structure so that it is processed correctly. A similar situation arises in handling persistent objects. In fact, a multiple-valued variable may contain objects of different types, as mentioned earlier. To allow the dynamic checking of object types, the attribute 'Class' is defined for all objects. The attribute returns the OID of the class of which the object is an instance. See Fig.3 for an example.

Jasmine/C balances static type reasoning and dynamic type checking. Static type reasoning is useful for restricting the search space of associative retrieval. Dynamic type checking is needed for multiple-type characteristics of Jasmine/C object variables.

4. Recursive Queries

Lack of recursive query capability has been considered as a drawback of relational database systems. The deductive database, combining logical reasoning with the relational data model, was proposed to extend the model to allow recursive query formulation [BRYF89]. Enhancement of SQL to allow recursive queries has also been proposed. We followed a third approach to recursive queries in designing Jasmine/C.

The basic idea is to utilize the recursive function call mechanism of the C language. The advantage of this approach is that new notation for writing recursive queries is unnecessary; C programmers can write recursive queries the same way they write recursive functions.

The procedural attributes Cost and Mass of COMPOSITEPART are recursive (see Fig.3 for Cost). This simply means that in each of the function bodies, a call to itself appears. In these programs, neither special operators nor new control structures for recursion are used.

We shall explain the object variable 'scan' and its related operations, although they do not have to do with recursion. They are necessary for retrieving one member of a set of objects. They are similar to the scan of System R [ASTR76] in concept, but are different in implementation. Scan objects are instances of a system-defined class SCAN. The class is created when a Jasmine/C program starts and is deleted when the program ends. As such, the class is temporary. The procedural attribute 'Openscan() ', defined on class OBJECT, returns a scan object. 'Next()' advances the scan.

5. Integrity and Annotation

Jasmine/C allows the annotation of attributes by demons, so that users may write procedures for maintaining integrity of the database.

There are two types of demon: before-demons

and after-demons. The before-demon is invoked before a specified action, such as an update, is made. The after-demon is invoked after the action. Not only properties, but also procedural attributes may be annotated.

Annotation of properties

Constraint, If-needed, If-updated, If-removed, and If-added are demons which are used to annotate properties. The Constraint demon, a before-demon, checks the validity of new values. Consider the annotation of Pno of class PART in Fig. 2, and the procedure in Fig. 6. The annotation says that Pno must be unique. (The unique Pno is introduced in the database for human communication. OID, although unique, is not appropriate for this.) The Constraint demon checks the uniqueness of the given number, and returns TRUE or FALSE depending on the result. Attempts to change the value fail if FALSE is returned. The variable 'value' is used to store the new value. The variable is multiple-valued when the annotated property is multiple-valued.

```
constraint {
  PART parts multiple;
  parts=PART where PART.Pno == value;
  if (parts == {} ) /* null set? */
    return(TRUE);
    else return(FALSE); }
```

Fig.6 The constraint on attribute Pno. The given value must be new.

The If-updated, If-removed, and If-added demons monitor the value of the property, and are invoked when the value is updated, removed, or added, respectively. If-removed and If-added demons are associated only with multiple-valued properties. An If-needed demon annotating a property is invoked when the requested value is undefined (i.e., NIL).

```
property INTEGER Stock
  if updated {
    ORDER neworder; /* ORDER is a class */
    if (self.Class == <BASEPART>  /* BASEPART? */
       && self.Stock<=20 )
       { neworder=ORDER.BuyParts(self.Stock*1.2); }
    else
    if (self.Class == <COMPOSITEPART> /* COMPOSITEPART? */
       && self.Stock <= 30 )
       { neworder=ORDER.ManufactureParts(self.Stock*1.5); }
  }
```

Fig.7 Attribute Stock is defined on class PART. See Fig.2. The demon defined here becoms active when the Stock value is updated to be less than specified values.

Fig.7 shows a demon annotating property Stock defined on PART. (See Fig.2.) The demon monitors the Stock property of each object in PART. If the value is modified to be less than a certain quantity, an order is given, either to buy or to manufacture the part in question. In this example, we assume such orders are stored and managed under class ORDER in the database.

Annotation of procedural attributes

Procedural attributes are useful to maintain some types of integrity that must hold even after databases are changed. For example, the mass of a composite part changes whenever one of its components is changed. This relationship always holds because the mass of the composite part is computed from those of the components whenever needed. This is the case when the composition of the part (i.e., component parts and number of each part used) is modified. However, when a component is deleted for some reason, something has to be done to prevent reference to the ghost object. This is known as the referential integrity problem of the relational databases [DATE81]. In programming languages, a similar anomaly occurs when data referenced by a pointer is deleted.

```
instance_method Delete inherited
  after_demon uninheritable
    { SUPPLIER supplier, sppls multiple;
      SCAN scan;
      sppls=SUPPLIER where SUPPLIER.Supply==self;
      scan=sppls.Openscan();
      while(!(scan==VOID))  {
        supplier=scan.Next();
        supplier.Supply-=self;
        /* Removal of self from set Supply. */
       }
      scan.Close();
     }
```

Fig.8 The definition of the after-demon annotating method Delete of class BASEPART, as shown in Fig.2. The method is inherited. The annotation is not inheritable; it is valid only for class BASEPART.

In Jasmine/C, annotating procedural attributes with demons helps maintain the integrity of databases vulnerable to such anomalies. For example, when a basepart is to be deleted from class BASEPART, the object's OID must be removed from the value of attribute Supply (actually a set of OIDs) of the basepart suppliers. 'Delete', a procedural attribute defined on class REFERENTIAL, is used to delete any referential object. Consequently, in the definition of the procedure, behavior for a specific situation such as the one described above, cannot be programmed. A 'trick' is needed so that the generic Delete procedure can be applied adaptively to any situation. In the definition of class BASEPART in Fig. 2, an

after-demon annotates instance-method Delete, which is actually inherited from REFERENTIAL. The annotation is valid only for BASEPART, since the demon is 'uninheritable'. The demon is invoked after the basepart object receiving the Delete message is deleted (See Fig.8).

The concept of a demon is well known in AI, and demons are implemented in AI shell languages [STEF86]. On the other hand, in the field of database systems, the counterpart of the demon is the 'trigger'. But, to the best of our knowledge, there are no database systems which support it. The reason seems to be twofold. First, supporting triggers is costly. Second, the capabilities of computationally complete programming languages rather than relationally complete query languages are needed to support general integrity. The Jasmine database system could overcome the second obstacle by providing Jasmine/C as a tool with which to write demons. The first concerns the implementation of database systems, in general, and the implementation techniques of addressing persistent objects, in particular. This will be discussed in the next section.

6. Implementation Issues

Architecture

The architecture of Jasmine is shown in Fig.9. A Jasmine/C program is compiled by the Jasmine/C compiler. The compiler then translates it into a C program. During the compilation (that is, preprocessing), the compiler accesses the database for information needed for type cheking, transformation of incomplete queries, and code generation.

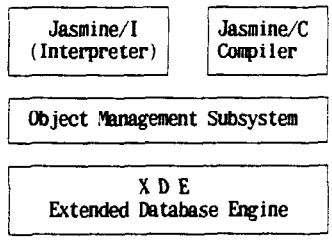

Fig.9 The Architecture of Jasmine.

Jasmine is implemented using XDE [YAMA89], an extended database engine that was developed for it. It runs under the UNIX ™ operating system.

XDE supports NF^2 tables (i.e., relations). The interface consists of C functions which implement join, selection, and other relational operations. Three kinds of table organization are supported: sequential, hash, and B-tree-like index. Indices on tables are not supported by XDE. They can be implemented using (possibly index-type) tables having one field containing tuple identifiers (TIDs). The TID is one of the field types recognized by XDE. Other field types are fixed-length-byte and variable-length-byte.

One consequence of using XDE is the reliance of Jasmine on value-based join operations instead of physical pointers, for not only set-oriented associative retrieval, but also for navigational access of objects. In fact, each persistent object is stored as one tuple in a table. The table forms the class of which the object is an instance. The tuple has a special field, the OID field, which contains a system-defined unique value labelling the object. Therefore, navigation from object to object involves either join operations or selections. The latter are used when an object is accessed via an object variable. Since the variable holds the OID, it is used to select the tuple having the same value in the OID field. When an index on the field is available, it is used preferentially.

Some advantages of this implementation are: (1) Because OID are independent of physical organization, linkages using OIDs between objects are not affected by any database reorganization, often needed in practical applications. (2) The meaning of object access using OIDs is clear and understandable. This helps users respond correctly when problems arise during navigation. Consider the difficulties programmers confront when an addressing exception occurs while manipulating pointers.

One disadvantage might be inefficiency. Direct access using TIDs or disc physical address could be an alternative. We believed that indexing on the OID field could shorten the access time.

Memory KB space

We introduced the memory KB (i.e., DB) space where users can create, destroy, and apply any relational operation to tables in the same manner as for disk-based tables. The memory KB space is used to keep temporal tables for temporal classes. Their lifetimes cannot be longer than that of the process which creates them. Tuples of a temporal table are addressed using memory addresses, so the access time is

far shorter than that of disk-based tables.

The introduction of memory KB space is justified when frame-based AI applications such as expert systems are considered. These applications are characterized as database intensive as well as computation intensive. They retrieve information from a large database and reason based on the information. During the reasoning process, they need to create, manipulate, and delete frames that store intermediate information. These frames do not have to be persistent.

Addressing tuples in the DB buffers

In XDE, a disk-based table is composed of pages whose size is either 4, 8,16, 32, or 64 kbytes. A tuple is addressed by its TID, which is derived from its logical page number and its offset within the page. To access a tuple in a buffer, first find the buffer storing the page. Second, compute the memory address of the tuple using the buffer address and the offset of the tuple. If the tuple is refered to several times in a program, it is possible to improve the performance by saving the memory address of the tuple when it is first accessed, and using the saved address for the following accesses. This idea is implemented for the single-valued 'self' variables in methods written in Jasmine/C. The variables retain the OID of the receiver object and its memory address, whenever the address is available.

The advantage of this implementation is that high-speed access to such a tuple becomes possible while allowing the application of the relational operations of XDE to the tuple. One of the demerits is that dynamic checking of the validity of the tuple address is necessary. Another is the necessity of fixing the buffer containing the tuple during program execution. This may require a large buffer area if the program refers to many persistent objects.

The last remark is connected with the reason why XDE supports NF^2 tables. The answer is simple: NF^2 tables are needed for multiple-valued properties. For example, Supply of class SUPPLIER is multiple-valued, so the value is a set of OIDs of baseparts. The value is stored in each tuple representing a supplier object. This natural clustering ensures higher speed for navigational and associative retrieval involving multiple-valued properties than does non-clustering in 1NF tables. Classes having attributes of type TUPLE are also mapped onto NF ^{2} tables. This is not discussed further because we are running out of space.

7. Conclusion

This paper described the main features and the design principles of the database programming language Jasmine/C. It is designed and implemented as an integrated part of the object-oriented database system Jasmine. Jasmine and Jasmine/C are interdependent; one cannot be used without the other.

In the design, many ideas were borrowed from and shared with other works: the functional data model with DAPLEX [SHIP81], IRIS [FISH87] [LYNG87], and GENESIS [BATO88], the object-oriented approach with GemStone [MAIE86], ORION [BANE87] [KIMW87], and O2 [LECL88], and the database programming concepts of VBASE [ANDR87] and O++ [AGRA89]. Here, we limit our discussion to works that are closely related to our work.

IRIS is similar to Jasmine in its model and its architecture. Both are based on the functional data model, have type hierarchy, and allow attribute inheritance. One minor difference is that IRIS permits multiple inheritance. Another is that, whereas both systems are built on top of a relational database engine, Jasmine's engine, XDE, supports NF^2 tables but IRIS's does not. A major difference resides in their approaches to query languges. For Jasmine, we start with the C programming language, and extend it so that it can handle persistent objects in a relationally complete fashion. The language is used to write not only queries, but also methods, and application programs. This makes Jasmine computationally complete [ATKI89]. OSQL of IRIS, an extention of SQL, is provided. Foreign functions can be written in C and are called in the where-clauses of OSQL. However, IRIS does not provide a uniform but versatile database programming language such as Jasmine/C.

VBASE supports two languages: TDL, a schema definition language, and COP, a C-based operation description language. They are not unified, whereas Jasmine/C is a unified language. A more important distinction is that COP does not have associative retrieval capability. The basic way to handle objects in VBASE is in a one-object-at-a-time fashion.

O++ is a unified database programming language that is used to define and manipulate objects in Ode databases. The language is an

extension of C++. The distinguished feature of O++ is that persistency is a property of object instances. In Jasmine, classes are persistent. The set-processing constructs which O++ provides is a 'for' loop enabling the iterative processing of objects in clusters. This is also a one-object-at-a-time method. Jasmine/C supports multiple-valued variables, to each of which associatively retrieved objects are assigned. Any attribute (i.e., function) applicable to it can be applied collectively. Thus, the 'loop' mechanism is hidden in the semantics of the dot notation of Jasmine/C.

[ATKI89] gives a list of features that any object-oriented database system must have. The current Jasmine has, we believe, ten of the twelve mandatory features. Concurrency and recovery, features that the current version of Jasmine does not have, will be implemented in the second version. Jasmine/C, which is the topic of this paper, gives Jasmine computational completeness, the 7th feature in the list.

Acknowledgments

We wish to thank H. Ishikawa and M. Miyagishima for contribution to the design and implementation of an early version of Jasmine, and F. Kozakura for implementing part of XDE. Our special thanks go to J. Tanahashi, who supported our efforts from the management side. This work is supported in part as the AIST/MITI large-scale project on Interoperable Database Systems by NEDO (New Energy and Industrial Technology Development Organization).

References

[AGRA89] Agrawal,R. and Gehani,N.H. Rationale for the Design of Persistence and Q uery Processing Facilities in the Database Programming Language O++. Proc. 2nd Int. Workshop on Database Programming Languages, Oregon Coast, June 1989

[ANDR87] Andrews,T. and Harris,C. Combining Language and Database Advances in an Object-Oriented Development Environment. Proc. 2nd OOPSLA Conf. Orlando, FL., Oct. 1987, 430-440

[ASTR76] Astraham,M.M., et al. System R: Relational Approach to Database Management. ACM TODS, Vol.1, No.2, June 1976

[ATKI87] Atkinson,M.P. and Buneman,O.P. Types and Persistence in Database Programming Languages. ACM Computing Surveys, Vol. 19, No. 2, June 1987, 105- 190

[ATKI89] Atkinson,M.P., Bancilhon,F., DeWitt,D., Dittrich,K., Maier,D. and Zdonik,S. The Object-Oriented Database System Manifesto. Proc. 1st Int. Conf. on DOOD, Kyoto, Japan, December 1989, 40-57

[BANC87] Bancilhon,F, Briggs,T., Khoshafian,S.,and Valduriez,P. FAD, a Powerful and Simple Database Language. Proc. 13th Int. Conf. VLDB, Brighton, England, Sept. 1987, 97-105

[BANE87] Banerjee,J., Chou,H., Garza,J.F., Kim,W., Woelk,D., Ballou,N., and Kim,H. Data Model Issues for Object-Oriented Applications. ACM Trans. Office Info. Syst., Vol. 5, No. 1, Jan. 1987, 3-26

[BATO88] Batory,D., Leung,T.M., and Wise,T.E. Implementation Concepts for an Extensible Data Model and Data Language. ACM Trans. Database Syst., Vol. 13, No. 3, Sept. 1988, 231-262

[BLOO87] Bloom T. and Zdonik, S.B. Issues in the Design of Object-Oriented Database Programming Languages. Proc. 2nd OOPSLA Conf. Orlando, FL., 1987, 441-451

[BRYF89] Bry,F. Query Evaluation in Recursive Databases: Bottom-up and Top-Down Reconciled. Proc. 1st Int. Conf. on DOOD, Kyoto, Dec. 1989, 20-39

[CARE88] Carey,M.J., Dewitt,D.J., and Vandenberg,S.L. A Data Model and Query Language for EXODUS. Proc. 1988 SIGMOD Conf., Chicago, IL., June 1988, 413-423

[COPE84] Copeland,G. and Maier,D. Making Smalltalk a Database System. Proc. 1984 SIGMOD Conf., June 1984, 316-325

[DATE81] Date,C.J. Referential Integrity. Proc.7th VLDB Conf., Cannes, France, Sept. 1981, 2-12

[FIKE85] Fikes,R. and Kehler,T. The role of Frame-based Representation in Reasoning. C.ACM, Vol. 28 No. 9, 904-920

[FISH87] Fishman,D.H., Beech,D., Cate,H.P., Chow,T., Connors,T., Davis,J.W., Derrett,N., Hoch,C.G., Kent,W.,

Lyngbeak,P., Mahbod,B., Neimat,M.A., Ryan,T.A., and Shan,M.C. Iris: An Object-Oriented Database Management System. ACM Trans. Office Info. Syst. Vol. 5, No. 1, Jan. 1987, 48-69

[GOLD83] Goldberg,A. and Robson,D. SMALLTALK-80: The Language and its Implementation. Addison-Wesley, Reading, Mass., 1983

[KERN78] Kernigham,B.W. and Richie,D.M. The C Programming Language. Prentice-Hall, London, 1978

[KIMW87] Kim,W., Chou,H., and Banerjee,J. Operations and Implementation of Complex Objects. Proc. 1987 IEEE Data Engineering Conf., Los Alamitos, CA., 1987, 626-633

[LECL88] Lecluse,C.,Richard,P., and Velez,F. O_2, an Object-Oriented Data Model. Proc. 1988 SIGMOD Conf., Chicago, IL., June 1988, 424-433

[LYNG87] Lyngbeak,P. and Vianu,V. Mapping a Semantic Database Model to the Relational Model. Proc. 1987 SIGMOD Conf., San Francisco, 1987, 132-142

[MAIE86] Maier,D., Stein,J., Otis,A, and Purdy,A. Development of an Object-Oriented DBMS. Proc. 1st OOPSLA Conf., Portland, OR., 1986, 472-484

[MINS75] Minsky,M. A Framework for Representing Knowledge. In 'The Psychology of Computer Vision', McGraw-Hill, Editted by Winston,P.H., 1975

[MORG83] Morgenstern,M. Active Databases as a Paradigm for Enhanced Computing Environments. Proc. 9th VLDB Conf., Florence, Italy, Oct. 1983

[ROTH88] Roth,M.A., Korth,H.F., and Silverschatz,A. Extended Algebra and Calculus for Nested Relational Databases. ACM Trans. Database Syst. Vol. 13, No. 4, Dec. 1988. 389-417

[SHIP81] Shipman,D.W. The Functional Data Model and the Data Language DAPLEX. ACM Trans. on Database Systems, Vol. 6 No. 1, March 1981, 140-173

[STEF84] Stefik,M. and Bobrow,D.G. Object-Oriented Programming: Themes and Variations. The AI Magagine, Vol.6 No.4, 40-62

[STEF86] Stefik,M., Bobrow,D.G., and Kahn,K.M. Integrating Access-Oriented Programming into a Multiparadigm Environment. IEEE Software, Vol. 13 No. 1, Jan. 1986, 10-18

[YAMA89] Yamane,Y. Narita,M., Kozakura,F., and Makinouchi,A. Design and Evaluation of a High Speed Extended Relational Database Engine, XRDB. Proc. 1st Int. Symposium on DASFAA, Seoul, Korea, April 1989, 52-60

[ZANI83] Zaniolo,C. The Database Language GEM. Proc. 1983 SIGMOD Int. Conf., May 1983