An object-relationship diagrammatic technique for object-oriented database definitions.

Introduction

The object-oriented (OO) approach to database management evolved from the object-oriented approach to programming [1, 5, 6], and is finding use in such application areas as software engineering, document preparation and management, and in the design and production of engineering parts. Typically OO database systems involve a built-in object-oriented programming language that is necessarily procedural [2, 4, 10, 11, 12, 13, 14], unlike non procedural relational database manipulation languages like SQL [9].

A consistent feature of the object-oriented approach is that associated with every object representation, which crudely corresponds to a tuple in the relational approach [8, 9], is a system generated identifier together with both lists of references, and individual references, to the object identifiers of related objects.

In an OO database, the relationships can be one-to-many, binary many-to-many, ternary many-to-many, recursive many-to-many, and ISA one-to-one relationships, as with relational databases,. However, unlike relational databases, these relationships are defined by the supporting individual and lists of reference, and are all included in the conceptual database definition. Furthermore, the reference attributes supporting a relationship in the object-oriented database definition are two-way.

All this, despite the utility of the approach for many applications, does introduce a level of complexity in the database definition that is absent from the relational approach, at least as far as relationships are concerned. An undesirable consequence is that an OO database definition does not easily communicate the structure of the database. As a result, it can be difficult for a programmer attempting to manipulate an OO database to keep all the objects, their relationships, and supporting individual and lists of references in mind. Consequently, there is a need for an easily interpreted diagrammatic approach to OO database definitions that is relatively rich - from a semantic viewpoint. In other words, database definition diagrams should be able to capture a great deal of the database semantics, particularly with respect to relationships.

Currently, entity-relationship (E-R) diagrams are frequently used for communicating database structure [7]. However, because E-R diagrams do not explicitly incorporate the system generated identifiers and method of handling relationships that is unique to the OO approach, it can be criticized as being semantically poor - at least in the context of OO databases.

The diagrammatic technique presented in this paper, or **object-relationship** (O-R) diagrammatic technique, is designed only for final database designs with the OO approach, and appears to satisfy the requirements of relationship-semantic richness and ease of interpretation in the OO context. This object-relationship technique has its origins in the extended Bachman diagrams that are fairly commonly used with relational databases in the final design

stage. Extended Bachman diagrams, originally proposed in [3], give explicit information about how relationships are supported, unlike E-R diagrams.

Extended Bachman diagrams were first introduced in the late 1970s with an experimental data model that attempted to combine the symplicity of relations with the owner-coupled set relationship concept of CODASYL [3]. The strong point of extended Bachman diagrams is that they capture more of the explicit support mechanism for relationships than most other types of database definition diagrams. Since the OO approach involves very explicit and quite complex support methods for relationships, this type of diagram, suitably modified, is a good candidate for OO database definition diagrams. The modifications are quite extensive however, so that the object-relationship diagrams presented in this paper do not look much like extended Bachman diagrams.

It is worth emphasizing that the object-relationship diagrams are not intended to replace the E-R diagrams commonly used in the initial high level design stage of OO database. Object-relationship diagrams are rather intended for ease of communicating the final OO database design, and for use by programmers when writing database manipulation programs.

Relational version of the project database

The project database chosen to illustrate the ideas behind object-relationship diagrams concerns document management. A relational version of the database is presented first, as this will permit

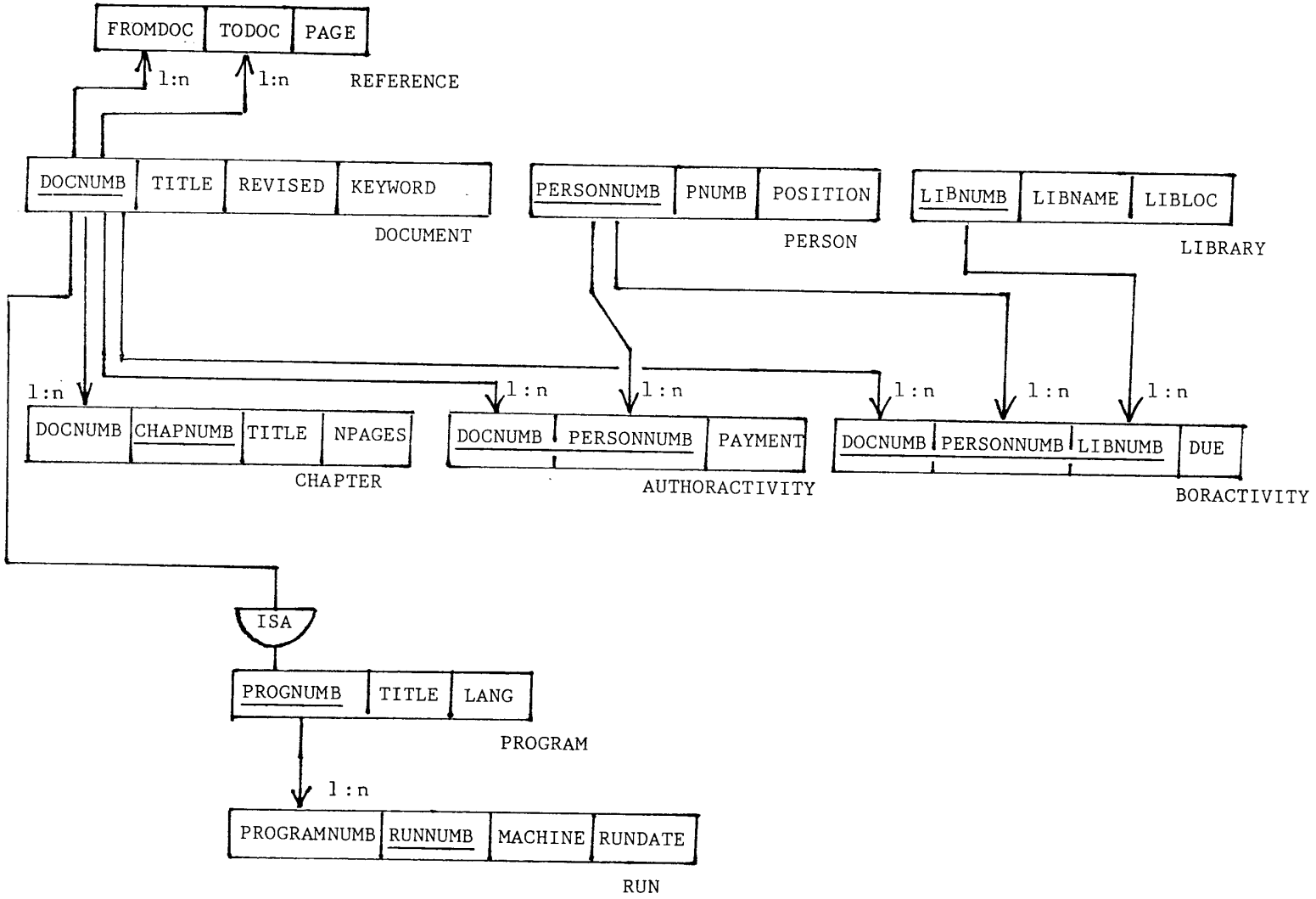
FROMDOC
TODOC
PAGE
REFERENCE
1:n
1:n
DOCNUMB
TITLE
REVISED
KEYWORD
DOCUMENT
PERSONNUMB
PNUMB
POSITION
PERSON
LIBNUMB
LIBNAME
LIBLOC
LIBRARY
DOCNUMB
CHAPNUMB
TITLE
NPAGES
CHAPTER
DOCNUMB
PERSONNUMB
PAYMENT
AUTHORACTIVITY
DOCNUMB
PERSONNUMB
LIBNUMB
DUE
BORACTIVITY
ISA
PROGNUMB
TITLE
LANG
PROGRAM
PROGRAMNUMB
RUNNUMB
MACHINE
RUNDATE
RUN

Figure 1

easy grasp of the semantics of the database, before the OO version need be discussed. The base tables for the database are illustrated by the extended Bachman diagram in Figure 1.

In an extended Bachman diagrams, a rectangular box is used for each relation, whether that relation represents a thing or a relationship, and the relation box is a string of further boxes that shows the primary key (underscored) and the attributes of the relation. A one-to-many (1:n) relationship is shown as an arrow from the primary key (or sometimes a candidate key) of the parent relation to the foreign key of the child relation. Many-to-many relationships are shown as composed of 1:n relationships, consistent with the way they are modeled in a relational database. Subtypes, taking part in IS-A-TYPE-OF or ISA relationships are shown as a line from the primary key of the supertype to the primary key of the subtype, with the line containing a bowl to symbolize the epsilon set inclusion symbol. As a result, E-B diagrams enable the relationship structure of a relational database to be evident at a glance, with the primary and foreign keys supporting any relationship being equally self-evident.

The database in Figure 1 contains all the types of relations that are encountered in OO databases. It is an extended and modified version of a database used by Cattell in a discussion of OO databases [6]. It has one-to-many relationships as well as a binary many-to-many, a ternary many-to-many, a recursive many-to-many, and a subtype (ISA) relationship, this last relationship permitting utilization of the inheritance concept in OO databases.

The main relation is DOCUMENT, each tuple of which describes a document. A document can have many chapters, with each

chapter represented by a tuple of the relation CHAPTER. A person can be both an author and a borrower of a document. A person can author many documents and a document can be authored by many persons. A person is represented by a tuple in PERSON and relation AUTHORACTIVITY enables the resulting many-to-many relationship between PERSON and DOCUMENT.

A document can have copies in many libraries, and can be checked out of a library by a person. A tuple in the relation LIBRARY represents a library, and a tuple in the relation BORACTIVITY represents a check-out of a document event from a library by a person. Thus BORACTIVITY enables the ternary many-to-many relationship between DOCUMENT, PERSON and LIBRARY.

A document contains references to other documents and to itself. A tuple in the relation REFERENCE describes a reference from a document (FROMDOC attribute) on a page number (PAGE attribute) to a document (TODOC attribute). Since a document can have many references there is a one-to-many relationship between DOCUMENT and REFERENCE supported by the foreign key FROMDOC. But, in addition, since a document can be referred to by many documents there is another one-to-many relationship between DOCUMENT and REFERENCE supported by the foreign key TODOC. These two 1:n relationships mean that DOCUMENT participates in a many-to-many relationship with itself, that is, in a recursive or cyclic many-to-many relationship. Thus there can be an explosion of references emanating from a single document: One document can reference a set of documents; each document of the set references a further set of documents, each of which references a further set of documents, and

so on. Similarly there can be an implosion of references. One document can be referred to by a set of documents, each of which can be referred to by a further set of documents, and so on.

Each tuple of the relation PROGRAM describes a computer program. Since a program is a kind of document, the set of PROGNUMB primary keys in PROGRAM are to be found among the set of DOCNUMB primary keys in DOCUMENT. Thus the relations DOCUMENT and PROGRAM form a subtype hierarchy, the relationship between PROGRAM and DOCUMENT being an ISA relationship. A tuple of RUN describes an execution of a program in PROGRAM. Since a program can be executed many times there is a one-to-many relationship between PROGRAM and RUN.

Note the significance of the ISA relationship in the database. Because of this relationship a tuple in PROGRAM inherits not only the attributes of the corresponding DOCUMENT tuple but also each relationship in which that DOCUMENT tuple participates. Thus we can have both legitimate requests involving attribute inheritance, such as:

"What are the titles of programs executed more than 100 times?"

```
SELECT TITLE FROM DOCUMENT, PROGRAM
WHERE DOCUMENT.DOCNUMB = PROGRAM.PROGNUMB
AND 100 < (SELECT COUNT(*) FROM RUN
           WHERE RUN.PROGRAMNUMB = PROGRAM.PROGNUMB);
```

and legitimate requests involving relationship inheritance:

"Who are the authors of C programs that have never been executed?"

```
SELECT PERSON# FROM PERSON
WHERE PERSONNUMB IN (SELECT PERSONNUMB FROM AUTHORACTIVITY
        WHERE DOCNUMB IN (SELECT PROGNUMB FROM PROGRAM
            WHERE LANG = "C" AND PROGNUMB NOT IN
                        (SELECT PROGRAMNUMB FROM RUN)));
```

The names of the attributes in Figure 1 were chosen to make the semantics self-apparent. Where this is not so, more detailed discussion later in the paper should clarify matters.

OO Data Models

An OO data model, like the relational data model, consists of the conceptual level and external level data structures permitted plus the operations neccessary to manipulate them [1]. However, unlike the relational model, in the OO approach many of the manipulation operations are so restricted that they are encapusulated with the conceptual-level data structures, which then, in essence, become abstract data types.

A diagram of the conceptual level of a specific database is most useful for depicting the specific conceptual level data structures involved, and not the (very restricted in the case of OO databases) processing to which those data structure might be sub-

jected. Nevertheless, although most would probably agree that a diagrammatic technique for OO databases should be able to easily communicate the database structure, it would sometimes be useful if it could include the routines that were encapsulated with individual objects. Nevertheless, this aspect of the matter is not addressed in this paper, for two reasons. First, it is a trivial matter, to diagrammatically enclose the representation for an object (including subtypes, if any) in a box that includes the encapsulated routines, and second, such enclosures, no matter what the diagrammatic technique, will severely clutter the diagram. A better way is probably to use simple structure diagrams to show encapsulated routines. Consequently, the object-relationship diagrammatic techniques proposed in this paper will be directed to the data structures used in conceptual-level OO databases, and will not be concerned with encapsulated routines.

Objects, attributes, object identifiers and primary keys

In the OO approach, an object has a unique identity that is independent of any values it contains [1, 6]. An object normally has associated attributes (sometimes called "instance variables"), and, as in the relational approach, one of these attributes, or a group of them, may be regarded as a primary key. However, because every object has a unique identity, an object need not have a primary key. Instead, the database system will generate a unique object identifier, which may or may not be accessible by the user, depending on the database system employed.

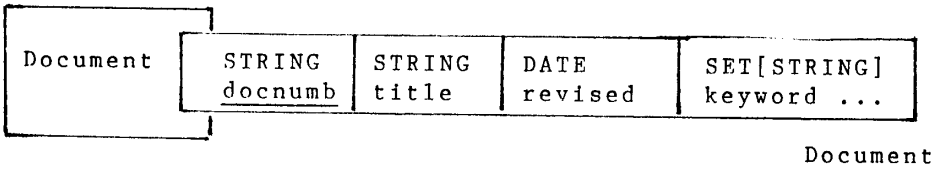

Figure 2a

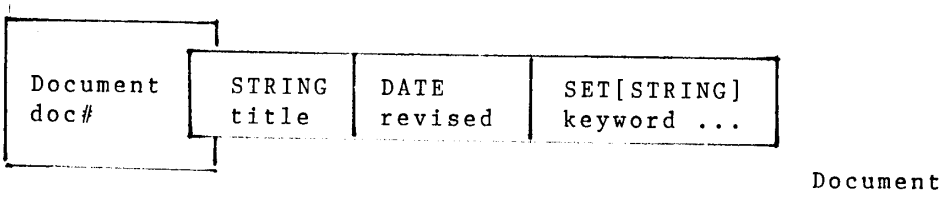

Figure 2b

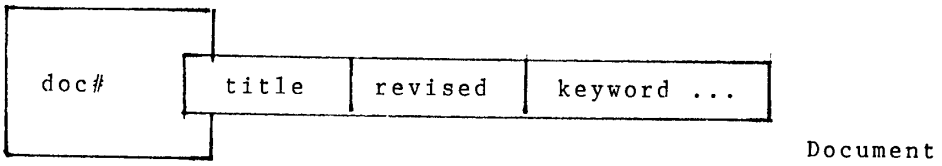

Figure 2c

As well as a possible primary key attribute, an object may have either simple attributes, such as a quantity or a name, whose type allows either literal numeric or alphnumeric values, or other defined values such as the type DATE , indicating a date value. An object may also have collection attributes, such as sets or lists, for example, the set of keywords in a document. The relational approach does not allow collection attributes.

These points can be illustrated by the diagrams in Figure 2a, 2b and 2c for **Document** objects. The conceptual database definition is shown in Figure 2d.

```
Document:  <
    doc#:        Document;
    title:       STRING;
    revised:     DATE;
    keyword:     SET[STRING];  >
```

Figure 2d

The version of the object **Document** where the primary key (**docnumb**, underscored) is included is shown in Figure 2a. The object name begins with an upper case character, attributes with lower case, and types are in upper case immediately above the attribute names. In this case no name is given for the internally generated object identifier "attribute", but its type is **Document**. Furthermore, the object identifier, which will normally not be visible to users, is shown as a larger rectangle attached to the rectangle for the other

attributes. This larger rectangle thus serves as a visual symbol for the unique identifier attribute of the object. The collective attribute **keyword** is denoted by the attribute name followed by ellipsis.

Because of the existance of an object identifier for each object, the primary key attribute can always be omitted, unless needed by users for non system purposes. This is done in the version in Figure 2b. In addition, the lack of a name for the object identifier attribute is remedied by using the attribute **doc#**, which therefore must have the type **Document**. Finally, the version in Figure 2c omits the attribute types, and shows just the attribute names, including the system-generated object identifier **doc#**, enclosed in a large rectangle.

Any of these versions could be used. We envision the third version (Figure 2c) as the most practical in the design stage. The advantage is that the named object identifier attribute, even if system generated and not normally accessible, is easier to handle than one with no name. It is not easy to discuss or communicate designs involving things with no names. Accordingly, we will proceed with objects diagrammed as in Figure 2c. In more complex diagrams, for clarity, this large rectangle for the object identifier can also be shaded (see Figure 3a, for example)

One-to-many relationships

In handling relationships, a simple diagrammatic principle is used: **A list attribute recessed in, but protruding, from the**

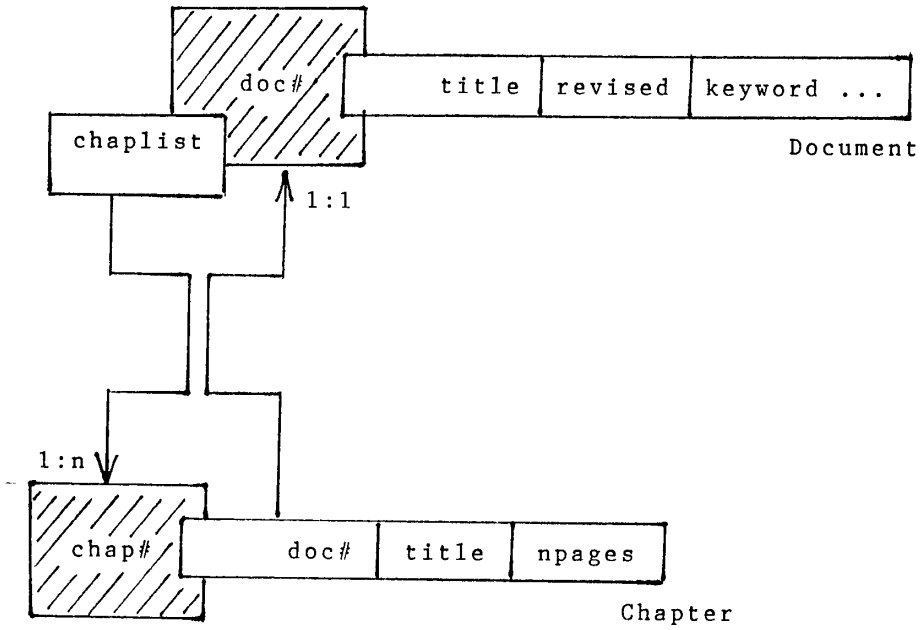
doc#
title
revised
keyword ...
Document
chaplist
1:1
1:n
chap#
doc#
title
npages
Chapter

Figure 3a

shaded rectangle representing the internally generated object identifier is a list of references to related objects, that is, a collective attribute of type LIST[Object], as in Figure 3a.

In Figure 3 there is an object-relationship diagram for the 1:n relationship between **Document** and **Chapter** objects, as based on the database definition in Figure 3b.

```
Document:  <
    doc#:       Document;
    title:      STRING;
    revised:    DATE;
    keyword:    SET[STRING];
    chaplist:   LIST[Chapter]; >

Chapter:   <
    chap#:      Chapter;
    doc#:       Document;
    title:      STRING;
    npages:     INTEGER; >
```

Figure 3b

In the object **Chapter**, the attribute **chap#** is taken as naming the object identifier for a chapter of a document. Accordingly, **chaplist**, which is a list of **chap#** values, is just one additional collective attribute of the object **Document**, but placed, according to the basic diagram rule, protruding from the object identifier

attribute in the object **Document**. Thus a **Document** object has a list, among its other attributes, of the object identifiers of its chapters. The type of **chaplist** must be LIST[**Chapter**]. Furthermore, in a **Chapter** object, there is an attribute **doc#** with the type **Document**, that is, its value must be a **Document** object identifier. These reference atributes define the 1:n relationship between the objects **Document** and **Chapter.**

In the diagram, an arrow, with a 1:n label, from the **chaplist** attribute in the parent **Document** object, to the object identifier attribute **chap#** in the child **Chapter** object, defines the reference pathway from the one parent to the many children of the 1:n relationship. Conversely, the return reference pathway from a single child to its parent is denoted by an arrow, with a 1:1 label, from the **doc#** attribute (a kind of foreign key) in the **Chapter** attribute to the object identifier attribute **doc#** in the **Document** object.

Since the two pathways of a 1:n relationship are actively used in the OO approach, they must both be clearly diagrammed. Unfortunately, in a diagram for a complex database, this can lead to a lot of arrows. In order to minimize relationship arrow clutter in a complex diagram, the two arrows may merge along the bodies of the arrows, as shown in Figure 3a.

Binary many-to-many relationships

Referring first to the relational database structure in Figure 1b, in Figure 4 we show how to diagram the OO version of the many-to-many relationship between **Person** and **Document** objects. Protruding

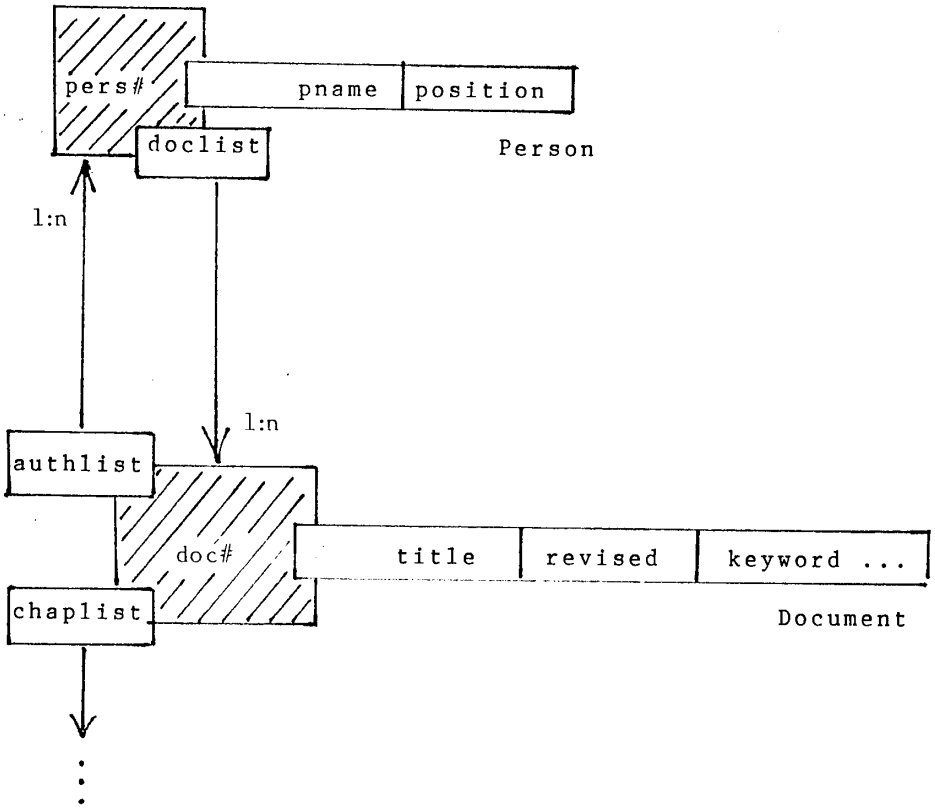
pers#
pname
position
doclist
Person
1:n
1:n
authlist
doc#
title
revised
keyword ...
chaplist
Document

Figure 4a

```
Document: <
    doc#:       Document;
    title:      STRING;
    revised:    DATE;
    keyword:    SET[STRING];
    chaplist:   LIST[Chapter];
    authlist:   LIST[Person];  >

Person: <
    pers#:      Person;
    doclist:    LIST[Document];
    pname:      STRING;
    position:   STRING;        >
```

Figure 4b

from the object identifier attribute **pers#** in the **Person** object is a LIST attribute **doclist**. In symetrical fashion, protruding from the object identifier **doc#** in the **Document** object is the LIST attribute **authlist**. The attribute **doclist** in **Person** gives the object identifiers of the documents authored by the **Person** object, and attribute **authlist** in **Document**. These references define the many-to-

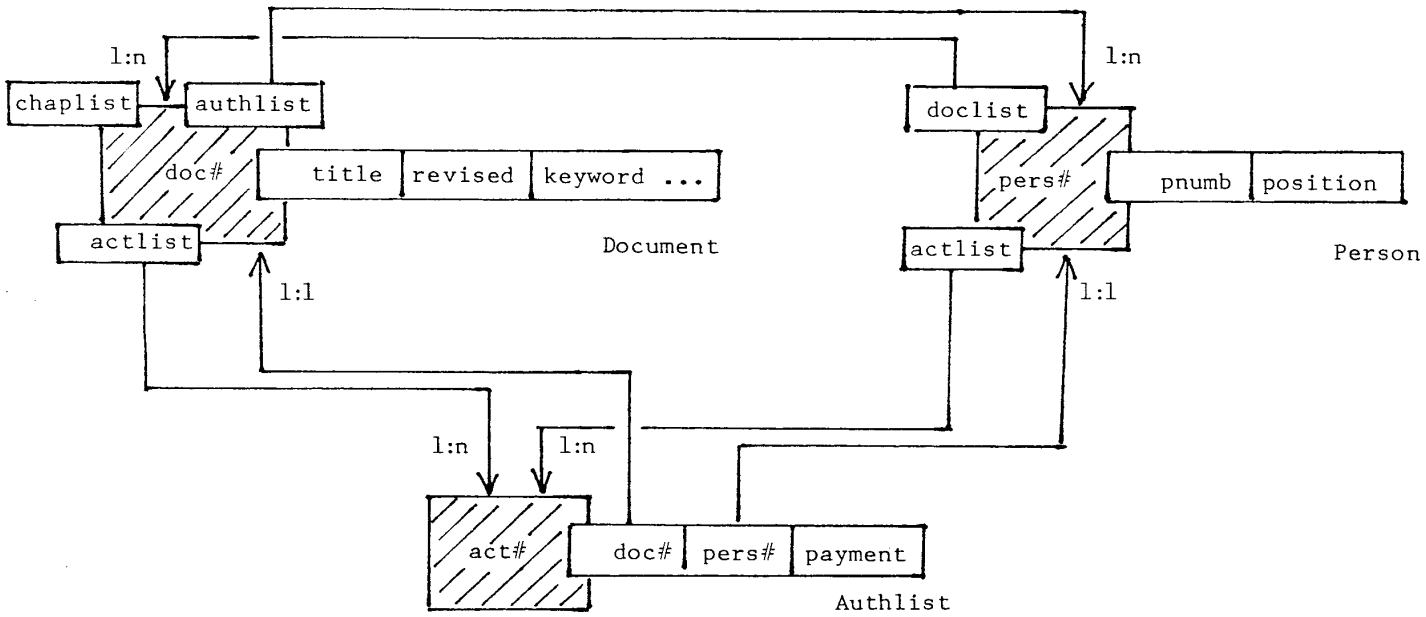
1:n
1:n
chaplist
authlist
doclist
doc#
title
revised
keyword ...
pers#
pnumb
position
actlist
Document
actlist
Person
1:1
1:1
1:n
1:n
act#
doc#
pers#
payment
Authlist

Figure 5a

many relationship between **Document** and **Person**, and are diagrammed by two arrows - one from **authlist** to **pers#** and one from **doclist to doc#.**

This method of handling a many-to-many relationship, using symmetrical reference lists, and only two objects, is different from the relational approach, which requires three relations, equivalent to three objects (Figure 1a). However, using only two objects is possible in the OO approach only in the less common case where there is no intersection data that are attributes of the pair of objects participating in the relationship, as is the case in Figure 4. But if it were the case, for example, that an author received a payment for each document that he or she contributed to, and this information was required, then a third object **Authact** is required, where the object is an authoring activity (see the relational version in Figure 1, where a corresponding relation like AUTHORACTIVITY is always necessary). This is handled as a pair of 1:n relationships, along the lines of the relational approach, but with reference lists to handle the 1:n relationships. This is illustrated in Figure 5.

```
Document:  <
    doc#:        Document;
    title:       STRING;
    revised:     DATE;
    keyword:     SET[STRING];
    chaplist:    LIST[Chapter];
    authlist:    LIST[Person];
```

```
    actlist:    LIST[Activity];  >

Authact:  <
    act#:       Authact;
    doc#:       Document;
    pers#:      Person;
    payment:    INTEGER; >


Person: <
    pers#:      Person;
    doclist:    LIST[Document];
    actlist:    LIST[Authact];
    pname:      STRING;
    position:   STRING;         >
```

Figure 5b

Notice that in Figure 5a the lists **doclist** and **authlist**, which enabled the many-to-many relationship in Figure 4 without intersection data, were not removed in this new version, although they could have been. By allowing them to remain the user has two ways of handling the many-to-many relationship. If the intersection data is of no interest, because of the inclusion of **doclist** and **authlist** the user can go directly from the **Document** entity to related **Persion** entities, and vice versa, and can thus group related **Person** data with a specific **Document** object, and vice versa. But in addi-

tion, because of **actlist** in both **Document** and **Person**, details of all the authoring activity for a given document can be retrieved, and details of all the document activity for a given author can be retrieved. This dual possibility of either using or ignoring the intersection data in a many-to-many relationship is found only in the 00 approach.

Ternary many-to-many relationships

Referring first to the relational database structure in Figure 1b, in Figure 6 we show how to diagram the 00 version of the ternary many-to-many relationship between **Person**, **Library** and **Document** objects. Unlike the case of a binary many-to-many relationship, with a ternary many-to-many relationship intersection data is necessary, and there has to be an object for the activity in which the objects **Person, Library** and **Document** participate. This is the borrowing activity object **Boractivity**, where an object is a borrowing event involving a library, a person and a document copy. Thus the relationship is handled in a manner similar to a binary many-to-many relationship, and is illustrated in Figure 6.

```
Document:  <
    doc#:       Document;
    title:      STRING;
    revised:    DATE;
    keyword:    SET[STRING];
    chaplist:   LIST[Chapter];
```

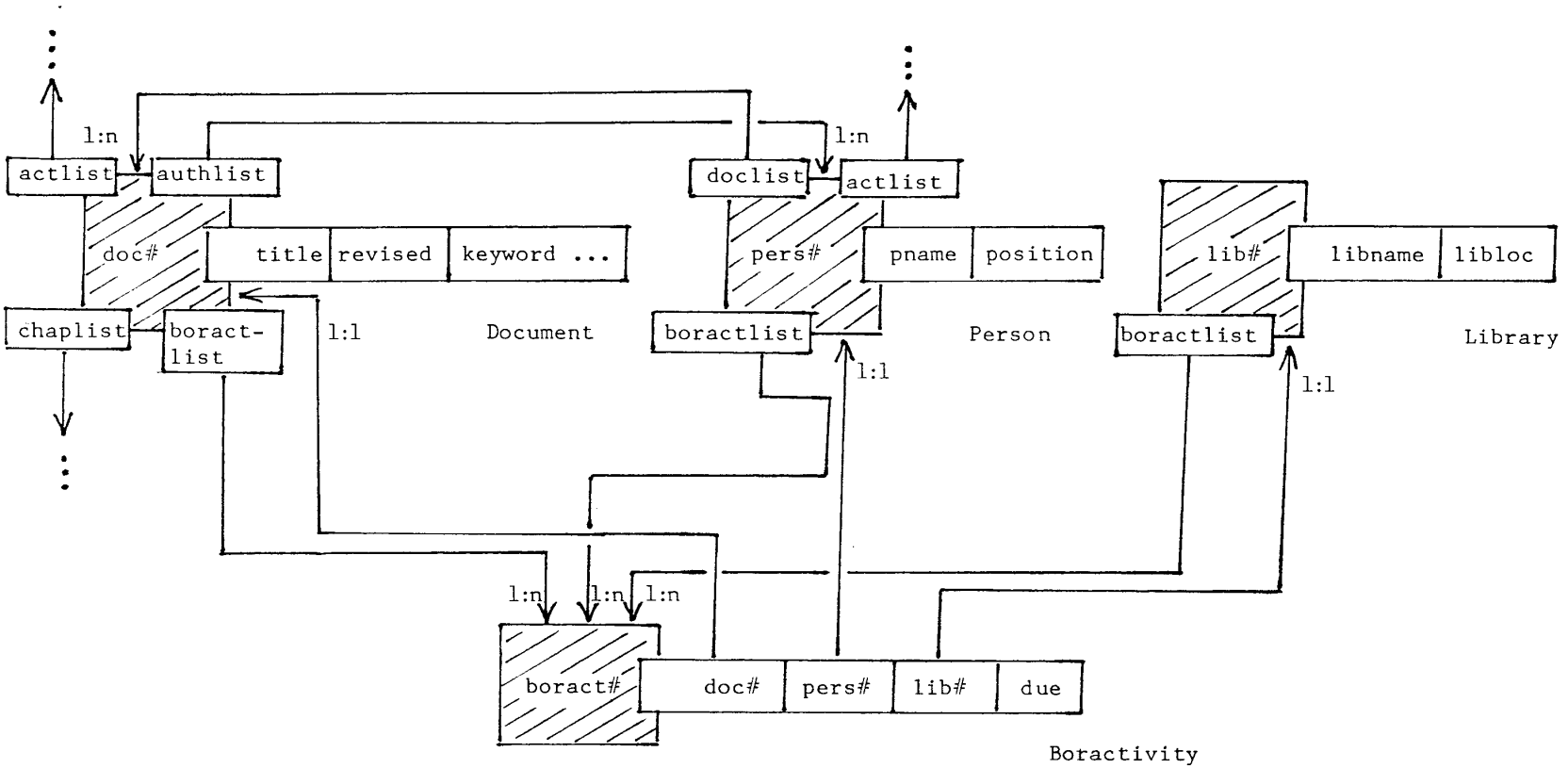
1:n
actlist
authlist
doc#
title
revised
keyword ...
chaplist
boract-
list
1:1
Document
1:n
doclist
actlist
pers#
pname
position
boractlist
1:1
Person
lib#
libname
libloc
boractlist
1:1
Library
1:n
1:n
1:n
boract#
doc#
pers#
lib#
due
Boractivity

Figure 6a

```
    authlist:    LIST[Person];
    actlist:     LIST[Activity];
    boractlist:  LIST[Boractivity];    >

Person: <
    pers#:       Person;
    doclist:     LIST[Document];
    actlist:     LIST[Activity];
    boractlist:  LIST[Boractivity];
    pname:       STRING;
    position:    STRING;          >

Library: <
    lib#:        Library;
    boractlist:  LIST[Boractivity];
    libname:     STRING;
    libloc:      STRING;   >

Boractivity:  <
    boract#:     Boractivity ;
    doc#:        Document;
    pers#:       Person;
    lib#:        Library;
    Due:         DATE;     >
```

Figure 6b

It is always possible to regard a ternary many-to-many relationship as consisting of three binary many-to-many relationships, in this case, between **Document** and **Library**, between **Library** and **Person** and between **Document** and **Person**. However, it is well known [9] that with most other database approaches, including the relational approach, if there is no primary key or equivalent that does not include the foreign keys, viewing a ternary relationship as three binary relationships is dangerous and may result in a loss of information. For example, if person p1 borows document d1 from library 11, and p1 also borrows d1 from library 13, then if we just consider the binary relationship between person and document, we have that p1 borowed d1, when in reality there are two distinct borrowings, one from 11 and the other from L2. However, if there is a unique event (event object) identifier, as will be the case in the OO approach, it will be clear that there were two separate borrowings even if only the binary many-to-many relationship between **Document** and **Person** is considered. This is conveyed by the diagram in Figure 6a, since there are two pathways (and twin arrows) for each of the 1:n relationships involving **Boractivity**.

Recursive many-to-many relationships

Referring to Figure 1, the relationship between the object **Document** and an object **Reference** is recursive (or cyclic) many-to-many, and breaks down into two distinct one-to-many relationships between **Document** and **Reference**. The recursive relationship can be regarded as a many-to-many relationship between **Document** and it-

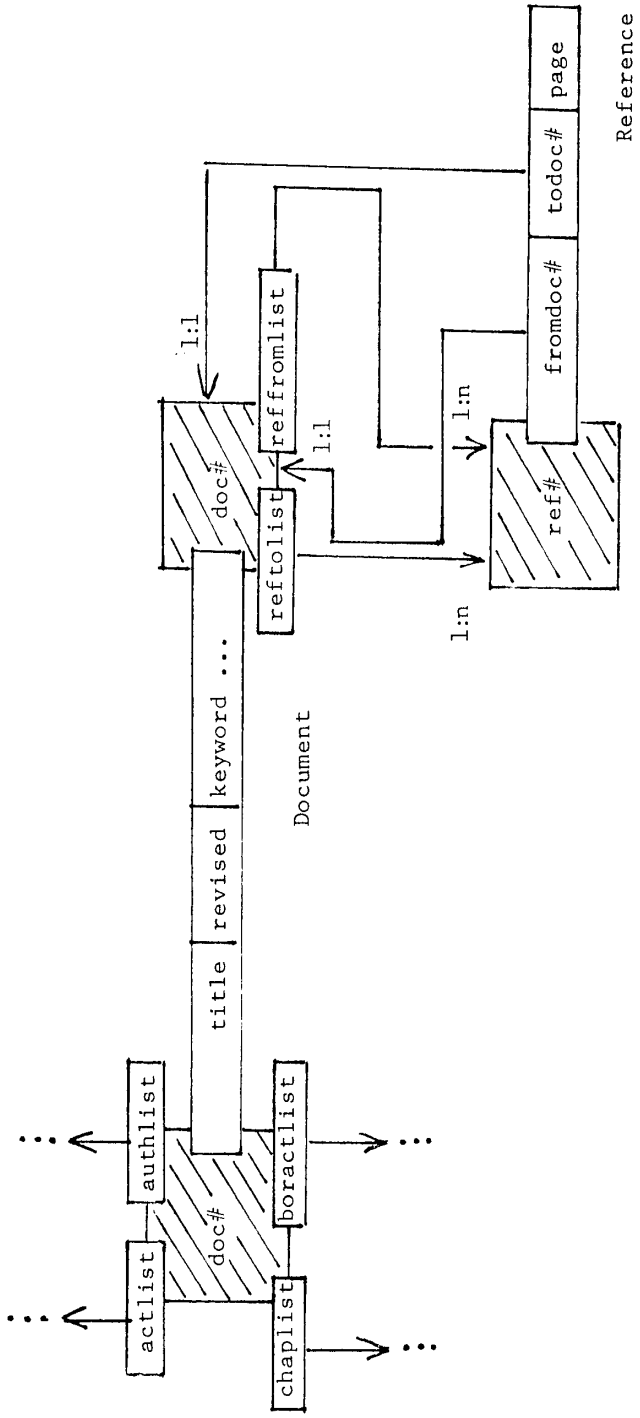
actlist
authlist
doc#
boractlist
chaplist
title
revised
keyword ...
Document
doc#
refolist
refromlist
1:1
1:n
1:1
1:n
ref#
fromdoc#
todoc#
page
Reference

Figure 7a

self, supported by an object **Reference** that contains intersection data. [Recursive many-to-many relationships always appear to require interesection data, although in theory a recursive many-to-many relationship is possible where interesection data is not required, the examples are not useful in practice.] Accordingly, the relationship can be diagrammed using the object-relationship diagram principles for each of the two 1:n relationships required. The object-relationship diagram is shown in Figure 7a and database definition is in Figure 7b.

```
Document: <
    doc#:        Document;
    title:       STRING;
    revised:     DATE;
    keyword:     SET[STRING];
    chaplist:     LIST[Chapter];
    authlist:     LIST[Person];
    actlist:      LIST[Activity];
    boractlist:  LIST[Boractivity];
    reftolist:    LIST[Reference];
    reffromlist: LIST[Reference];   >

Reference: <
    ref#:        Reference;
    fromdoc#:    Document;
    todoc#:      Document;
    page:        INTEGER;  >
```

Figure 7b

At this stage in the build-up of the attributes of **Document**, because the object **Document** has become involved in so many relationships, it contains many lists of identifers of related objects. When there are so many relationships involving a single object, in a diagram there will not be space for all of the lists of identifiers that protrude from the rectangle denoting the object identifier. The solution is to replicate the object identifier in the diagram for the object, as many times as are needed, as shown in Figure 7a for the case of **Document**. Incidently, this proliferation of reference list attributes, so obvious in Figure 7a, is completely avoided in the relational approach - by simply omitting it.

Note that **reftolist** in **Document** pairs with **fromdoc#** in **Reference** to support one of the two one-to-many relationships, and **refromlist** pairs with **todoc#** to support the other one. The diagram thus makes clear a state of affairs that frequently causes confusion and programmer errors. This problem does not occur in the relational approach.

Subtype one-to-one (ISA) relationships

We now show how a subtype or IAS relationship is be be diagrammed, recalling the DOCUMENT and PROGRAM relations from the database in Figure 1, a **Program** object is a kind of **Document** ob-

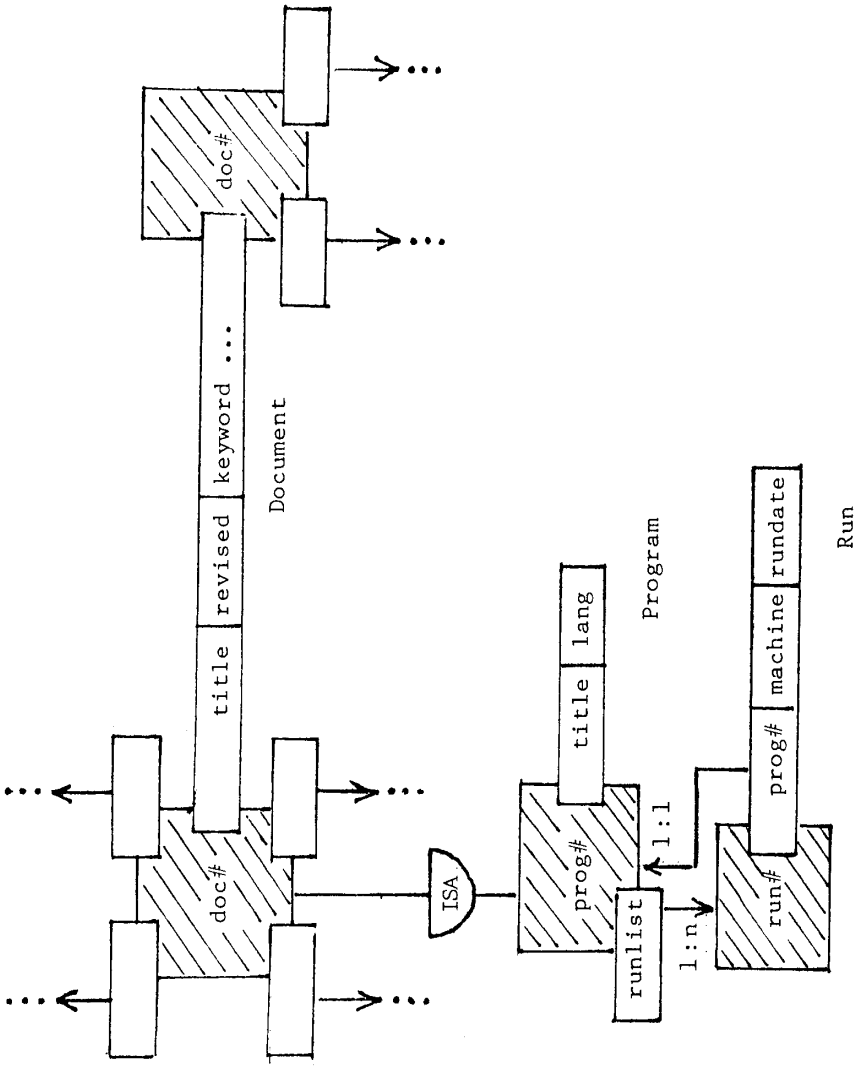
doc#
keyword ...
revised
title
Document
ISA
prog#
runlist
title
lang
Program
1:1
1:n
run#
prog#
machine
rundate
Run

Figure 8a

ject. Not all documents are programs but all programs are documents, so that there is a subtype 1:1, or ISA relationship between **Document** and **Program.** A program can have many executions, so that there is also a one-to-many relationship between **Program** and **Run.** The one-to-many relationship is handled in the usual way (Figure 2). The ISA relationship is dagrammed using an ISA cup, with the curved part of the cup symbolizing the epsilon set membership symbol from set theory (Figure 8a). Thus the diagram shows that a program is a type of document, and not the other way round. Not that with this relationship, since it is a kind of 1:1 relationship, no list of identifiers is needed in either participant. The database definition is in Figure 8b (reference lists needed in Document for other relationships are omitted to avoid clutter)

```
Document:  <
    doc#:        Document;
    title:       STRING;
    revised:     DATE;
    keyword:     SET[STRING];
    chaplist:    LIST[Chapter];
    authlist:    LIST[Person];
    actlist:     LIST[Activity];
    boractlist:  LIST[Boractivity];
    refto:       LIST[Reference];
    reffrom:     LIST[Reference];    >

Program: <
```

```
    prog#:      Document
    title:      STRING;
    lang:       STRING;
    runlist:    LIST[Run];  >

Run: <
    run#:       Run;
    prog#       Program;
    machine:    STRING;
    rundate:    DATE; >
```

Figure 8b

Summary

An object-relationship diagrammatic technique for capturing and communicating the semantics of OO databases has been presented and analysed. It is capable of display of objects, in terms of system generated object identifiers, simple attributes, collection attributes, and simple and collection reference attributes. The type of each attribute may be included in the diagram if required for displaying additional semantics. A one-to-many relationship may be diagrammed by means of a list of references, that is, a collection attribute of the parent object, together with a simple reference attribute of the child object; in addition, a pair or arrows indicates the two resulting reference pathways, one from parent to children, and the other from child to parent.

Two alternative methods of diagramming a binary many-to-many relationship are allowed, in accordance with the two possible ways such a relationship may be defined in the OO approach. One approach involves a third object with intersection data attributes and two one-to-many relationships. The other method involves no intersection data object, but a pair of arrows and symetrical lists of reference attributes, each arrow emanating from a list of references.

Ternary many-to-many relationships are handled like binary many-to-many relationships with interesection data. A recursive relationship is handled as a pair of one-to-many relationships. A simple diagrammatic symbol is used to denote a subtype (ISA) one-to-one relationship.

Object relationship diagrams are not meant to replace entity-relationship diagrams commonly used in the early stages of object-oriented database design. They are intended instead for capturing the semantics of final database design and ease of communication of database structure to programmers and others involved in using OO databases.

REFERENCES

1. Abiteboul, S., Hall, R. IFO, a formal semantic data base model, ACM Trans. on Database Systems, 12 (4), 1987.

2. Bancilhon, F., et al. The design and implementation of O_2, an object-oriented DBMS, in "Advances in Object Oriented Database Systems," K. R. Dittrich, ed., Computer Science Lecture Notes 334, Springer Verlag, New York, 1988.

3. Bradley, J., An extended owner-couple set data model and predicate calculus for data base management, ACM Trans. on Database Systems, 3(4), 1978, 385-416.

4. Bret, R., et al. The Gemstone Data Management System, in "Object-Oriented Concepts, Databases and Applications, W. Kim, F.H. Lochovsky, Eds., Addison-Wesley, Reading, Mass, 1988.

5. Cardenas, A. F., McLeod, D. "Research Foundations in Object-Oriented and Semantic Databases," Prentice Hall, Englewood Cliffs, New Jersey, 1990.

6. Cattel, R. G. G. "Object Data Management" . Addison Wesley, 1991.

7. Chen, P.P. The entity-relationship model: Towards a uniform view of data, ACM Trans. on Database Systems, 1 (1), 1976, 9-36.

8. Codd, E. F. Relational databases, a practical design for productivity, CACM, 25(2), 1982, 109-117.

9. Date, C. J. "Introduction to Database Systems, 5th ed., Addison Wesley, Reading, Mass., 1990.

10. Kim, W. et al. Features of the ORION object-oriented DBMS, in "Object-Oriented Concepts, Databases and Applications, W. Kim, F.H. Lochovsky, Eds., Addison-Wesley, Reading, Mass, 1988.

11. Object Design. ObjectStore Reference Manual, Object Design, Inc., Burlington, Mass., 1990

12. Objectivity. Objectivity Database Reference Manual, Objectivity Inc., Menlo Park, California, 1990.

13. Ontologic. ONTOS Reference Manual, Ontologic Inc., Billerica, Mass., 1989.

14. Versant Object Technology. Versant Reference Manual, Versant Object Technology Inc., Menlo Park, California, 1990.