THE DATA INDEPENDENT ACCESSING MODEL

A Review and Discussion

James O. Mathers

COINS Technical Report 74A-1

February 1974

Preface

The Data Independent Accessing Model was developed by M.E. Senko, E.B. Altman, M.M.Astrahan, P.L. Fehder, and C.P. Wang as part of the Universal Information Syatem Technology (UIST) Project at IBM Research Laboratory, San Jose, California. The Model is described in detail in IBM Research documents RJ982, A Universal Information System Technology 1, February 25, 1972, and RJ1121, The Representation Independent Language, (Parts 1,2,and 3), November 2, 1972. The work of this Project is also reported in an article by Senko, Altman, Astrahan, and Fehder, "Data Structures and Accessing in Data-base Systems", IBM Systems Journal, Volume 12, Number 1, 1973.

The purpose of this paper is to summarize in one document the main features of the Data Independent Accessing Model and to provide additional explanatory examples. Sections 1 through 3 are devoted to this review. In Sections 4 and 5 some areas of the Model that were not developed in the IBM Project are explored. Specifically, a Representation Dependent Language is proposed and its use in accessing data in the model is discussed.

CONTENTS

1 - The Data Independent Accessing Model

The Data Independent Accessing Model (DIAM) is an information storage scheme in which the factors relevant to data base design and use have been grouped into four major areas or sub-models. The grouping is done in such a way that decisions can be made within the context of one sub-model more or less independently of the structures of other sub-models. Most significantly this means that, given a set of data stored in the form of the DIAM, a user will be able to access any subset of the data without concern for its physical location or organization.

The four sub-models of the DIAM are:

Entity Set Model - Defines the user's interface with the stored information. Data is described as named sets of data items.

String Model - Defines the logical access paths among data items and collections of data items within the named sets.

Encoding Model - Specifies how logical data items and their access paths are represented as bit strings.

Physical Device Model - Specifies the form and arrangement of bit strings on secondary storage media.

The sub-models represent varying levels of abstraction of a given set of data. The structure of each model is specified by values of parameters that are appropriate to its level. For example, the Entity Set Model is specified by giving the name and content of the data sets that will be available to the user. At the Physical Device level, parameter values will be the names and sizes of areas on physical storage devices that actually hold the data.

In order to make use of a model to access and manipulate data items, it is necessary to be able to express the desired operations in terms of elements of the model. This implies the existence of a language whose primitive elements correspond to the elements of the model and the permissible operations on them. Potentially, each sub-model in the DIAM could have such a language associated with it, but at this time only two have been developed.

The language associated with the Entity Set Model is the Representation Independent Language (RIL) in which the primitive elements are sets of data, attribute values, and set manipulation operations. This is the language in which users can formulate their data base query requests.

The language associated with the String Model is the Representation Dependent Language (RDL) in which access paths within data sets are specified as well as attribute values. Typical operations in the RDL might involve creating or following access paths and retrieving attribute values.

Formal languages have not been developed to express data operations in terms of the Encoding or Physical Device models, but it is anticipated that these will take the form of collections of standard procedures designed to implement requests formulated at the more abstract levels.

Although the DIAM is partitioned for conceptual convenience, it is in reality one model and data is actually stored in only one form, namely that described by the Physical Device Model. Thus operations expressed in the language of any other model must be translated eventually into procedures that manipulate the physical representations of the data. To accomplish this, translation facilities are provided between the levels. Using them, a statement in the RIL will be converted to a set of statements in the RDL. This set will in turn be used to generate procedures at the encoding level which will finally be translated into physical device oriented procedures.

For example, a request for retrieval of a certain data value might be stated at the entity set level as the formation of a subset containing the desired item. At the string level the retrieval activity would appear as a command to follow the appropriate path to the data item. At the encoding level the path-following command would be translated into a sequence of pointer retrievals, since pointers are used to encode the paths. Finally, at the lowest level, pointers are mapped into addresses on a physical device.

The operations that may be performed within the DIAM include not only standard manipulations such as retrieval, update and deletion, but also alterations in the structure of the model. For instance, new data sets can be created or new access paths defined, using operations provided in the RIL and RDL. Because of this, model design activities can take place at a relatively high level and even the designer may take advantage of the independence of the sub-models as he can operate only on the phases of the model that are of interest. This facility permits the possibility of structure evolution or self-adaptation to changing query loads.

Figure 1-1 illustrates the relationships among the elements of the DIAM that have been described.

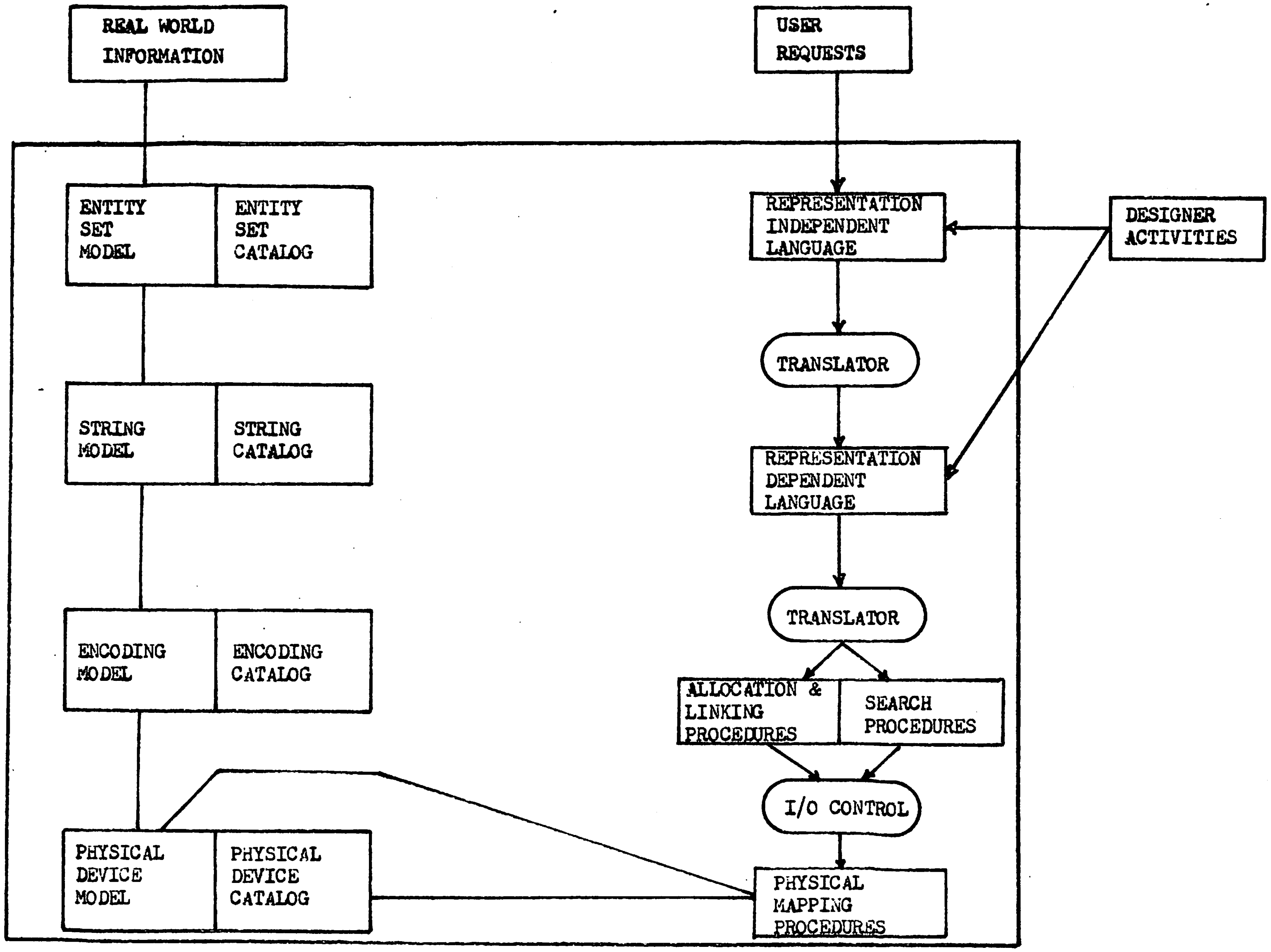

REAL WORLD INFORMATION
USER REQUESTS
ENTITY SET MODEL
ENTITY SET CATALOG
REPRESENTATION INDEPENDENT LANGUAGE
DESIGNER ACTIVITIES
TRANSLATOR
STRING MODEL
STRING CATALOG
REPRESENTATION DEPENDENT LANGUAGE
TRANSLATOR
ENCODING MODEL
ENCODING CATALOG
ALLOCATION & LINKING PROCEDURES
SEARCH PROCEDURES
I/O CONTROL
PHYSICAL DEVICE MODEL
PHYSICAL DEVICE CATALOG
PHYSICAL MAPPING PROCEDURES

FIGURE 1-1

1.1 - Entity Set Model

The Entity Set Model (ESM) provides the description of the stored data that is seen by the user. He will formulate his requests for retrieval, insertion, update, and modification of data items in terms of the structures of this model.

The basic item of information in the ESM is the entity, which corresponds to some real world object or concept about which information is being recorded. The entity appears in the ESM as a collection of attribute values which describe the object the entity is representing. These values are placed in the standard format of a triplet:

Attribute Domain Name/Role Name/Attribute Value

The attribute domain name is the set of entity names from which the attribute value can be drawn. The role name indicates the role the attribute value plays in describing the real world object. The attribute value is the name of the entity which, by its association with the real world object, describes it. The collection of triplets associated with a given entity is called its entity description and each triplet in the entity description must contain a unique role name. An entity description corresponds to a record in other systems.

To illustrate the concepts defined so far, suppose the real world concept about which we wish to record information is a student, and that his interesting features are his Student# (say 123) and his faculty advisor (say Mr.A). The entity corresponding to this student has a description consisting of the triplets:

$$\left\{\begin{array}{l}\text{NUMBERS/STUD\#/123}\\ \text{NAMES/ADVISOR/Mr.A}\end{array}\right.$$

where NUMBERS and NAMES are attribute domain names, STUD# and ADVISOR are role names and '123' and 'Mr.A' are attribute values.

If several objects have similarities that are of interest to the user, the entities which represent them may be grouped into an entity set. The descriptions of the entities in this set form an entity description set, which is roughly analogous to a file in other systems. Each entity description in the set will utilize the same role names in its triplets, and hence role names are unique across the description set.

If we were interested in data about several students we could have the following description set with an entity description for each student.

$$\left\{\begin{array}{l} \left\{\begin{array}{l}\text{NUMBERS/STUD\#/123}\\ \text{NAMES/ADVISOR/Mr.A}\end{array}\right. \\ \\ \left\{\begin{array}{l}\text{NUMBERS/STUD\#/456}\\ \text{NAMES/ADVISOR/Mr.B}\end{array}\right. \\ \\ \left\{\begin{array}{l}\text{NUMBERS/STUD\#/789}\\ \text{NAMES/ADVISOR/Mr.A}\end{array}\right. \end{array}\right.$$

We can give this description set a unique name, say STUDENTS, and then combinations such as Description Set Name.Role Name will be unique across the entire ESM. (e.g. STUDENTS.STUD#)

It should be noted that the attribute domains (such as NUMBERS) are themselves entity sets and in some contexts it may be of interest to provide each element with an entity description of its own. In the DIAM attribute domains are merely entity sets used for the special purpose of providing values to describe other entities. This use does not preclude the members of an attribute domain from being described by still other entities.

There must be at least one role name, or combination of several role names, which can never take on the same values in two different entity descriptions of the same description set. The role names with this property are the identifiers of the description set. In the description set STUDENTS, which was described above, the role name STUD# serves as an identifier.

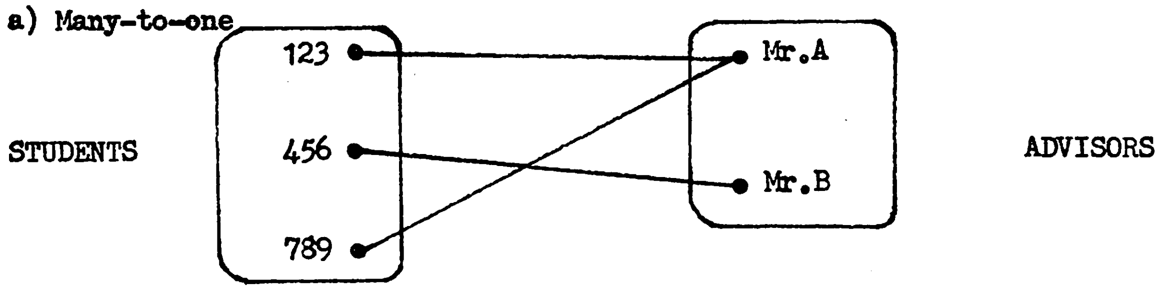
The construction of an ESM of information about a collection of real world objects is a process of building entity description sets by choosing the role names whose values will be the information of interest and of assigning these role names to description sets. But the way in which this assignment is made will have a great effect on the efficiency with which the user can manipulate the model to serve his purposes, and for this reason certain rules are provided to guide the creation of description sets. These are the Role Name Identifier Allocation (RNIA) rules.

Since attribute domains are actually entity sets, the values in an attribute domain identify unique entities. These may be used to describe other entities and may themselves be described. The problem is to decide: 1) What entity description sets will be formed and what role names will be the identifiers of each?, and 2) In which entity description sets will the non-identifier role names appear? For example suppose we wish to record the courses in which a student is enrolled and the instructor for for each course. STUDENTS, COURSES, and INSTRUCTORS are all entity sets. Course entities are used to describe students, and courses are in turn described by instructor entities. Given this information to be stored, we must choose among several possible models. There could be a single entity description set in which triplets containing course names and instructor names are part of entity descriptions for students, or STUDENTS could contain only the names of courses taken by each student with another description set, COURSES, containing the instructor information. Several other configurations are possible, and the RNIA rules will lead to the selection of the structure with certain desired properties.

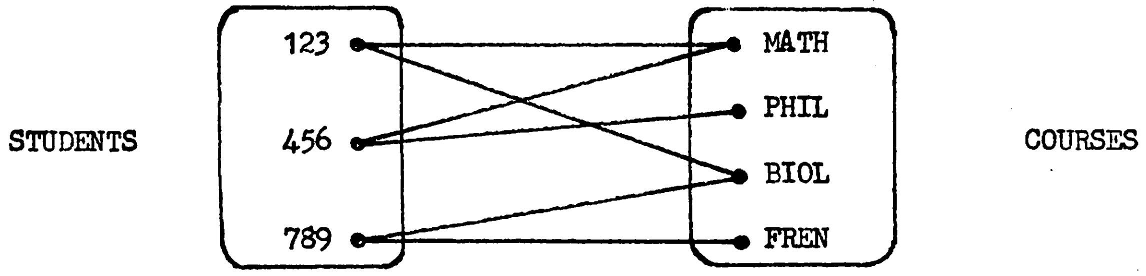
To apply the rules it will be necessary to examine the relationship that exists between the entity sets involved. For instance a student has only one faculty advisor but each advisor can have several students assigned to him, so the relation between the entity sets STUDENTS and ADVISORS is many-to-one. The relationship of STUDENTS to COURSES is obviously many-to-many. Figure 1-2 illustrates some possible relationships. The type of relation will determine the allocation of role names to descriptive sets.

FIGURE 1-2. Examples of Entity Set Relations

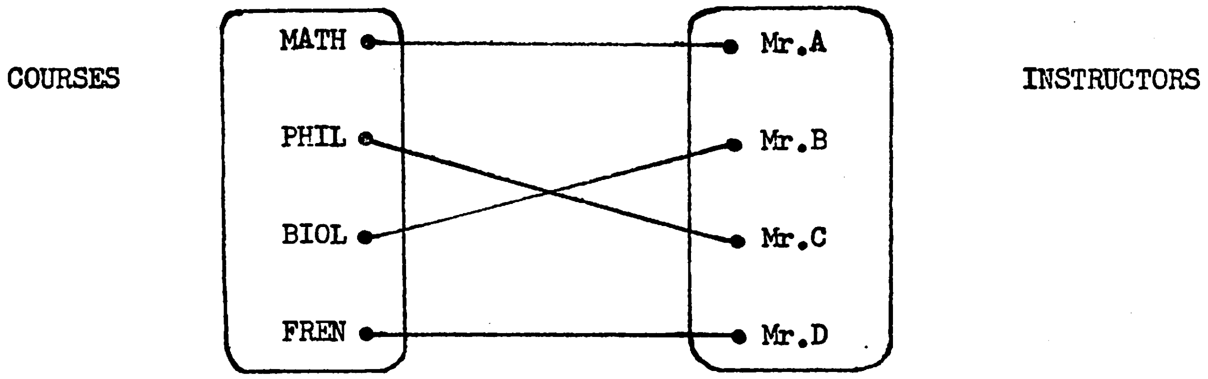
a) Many-to-one



b) Many-to-many



c) One-to-one



<u>RNIA Rule#1</u> - If the relation between two entity sets is many-to-one, an entity description should be formed for each of the "many" entities and each description should include a triplet in which the attribute value is a "one" entity. In the case of STUDENTS and ADVISORS this rule leads to a single description set in which STUD# is the identifier and the non-identifier role name ADVISOR appears in each description. The description set STUDENTS has been previously shown in this form.

<u>RNIA Rule#2</u> - If the relation between two entity sets is many-tomany, entity descriptions should be created to represent each association of entities from the two sets. These descriptions will contain at least two triplets whose values are the names of the two entities in the association. The role names in these triplets will be the identifiers of the description set. Figure 1-2b shows that a many-to-many relation exists between STUDENTS and COURSES and so to record information about what students take what courses Rule #2 requires the formation of a description set (call it SCHEDULE) containing the following entity descriptions:

SCHEDULE

NUMBERS/STUD#/123
COURSES/COURSE/MATH

NUMBERS/STUD#/123
COURSES/COURSE/BIOL

NUMBERS/STUD#/456
COURSES/COURSE/MATH

NUMBERS/STUD#/456
COURSES/COURSE/PHIL

NUMBERS/STUD#/789
COURSES/COURSE/BIOL

NUMBERS/STUD#/789
COURSES/COURSE/FREN

The identifiers for this description set are STUD# and COURSE.

Any information that further describes the association entities can be represented by additional triplets in the entity descriptions. For example if we wished to record the grade of a given student in a given course, each entity description in SCHEDULE would contain a triplet with the role name GRADE and a value drawn from the attribute domain GRADES.

RNIA Rule#3 - If two entity sets have a one-to-one relationship, the role names should be allocated according to the stability of the relationship over time. If associations between entities are likely to change relatively frequently, there should be a description set for each entity set and the associations should be shown by placing triplets naming entities from one set in the entity descriptions for the other set. Suppose we have the COURSE-INSTRUCTOR relation from Figure 1-2c and also wish to store the degree possessed by each instructor. The proper sets would be:

```
            { COURSES/COURSE/MATH                      { NAMES/INSTR/Mr.A
            { NAMES/INSTR/Mr.A                         { DEGREES/DEGREE/Ph.D.

            { COURSES/COURSE/PHIL                      { NAMES/INSTR/Mr.B
            { NAMES/INSTR/Mr.C                         { DEGREES/DEGREE/M.A.
COURSES                                  INSTRUCTORS
            { COURSES/COURSE/BIOL                      { NAMES/INSTR/Mr.C
            { NAMES/INSTR/Mr.B                         { DEGREES/DEGREE/M.S.

            { COURSES/COURSE/FREN                      { NAMES/INSTR/Mr.D
            { NAMES/INSTR/Mr.D                         { DEGREES/DEGREE/M.A.
```

This structure has the advantage that when the instructor for a course is changed the degree information is not affected as it would be if it were part of the entity description in COURSES.

If a one-to-one relation is fairly permanent, only one description set containing all the information will be needed since the updating problems will be infrequent.

It is interesting to note the relationships between the Entity Set Model and Codd's Relational Model. (E.F. Codd, "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, June, 1970, Volume 13, Number 6, page 377). An entity description set corresponds to a relation in the Relational Model and an entity description corresponds to an n-tuple of a relation. In addition, if an Entity Set Model for a given set of data is formed according to the RNIA rules and then transformed into a Relational Model by writing each description set in the form of a relation, the resulting group of relations will, in general, be in Third Normal Form.

We will conclude this discussion of the Entity Set Model by presenting the Description Set Catalog. Catalogs appear as part of each sub-model of the DIAM. Their purpose is to list in anorganized manner the elements of their respective sub-models. A catalog entry, consisting of an element name and certain associated information, constitutes a type specification for a class of element instances that make up a portion of the sub-model. The information given in a catalog entry is parameter information that is common to all instances of the given element type.

The Description Set Catalog (Figure 1-3) is the catalog for the Entity Set Model. The elements listed in it are either description set names (DSN) or role names (RN) and appropriate parameter information is given in each entry. The entry for STUDENTS means that the ESM contains a description set of the type STUDENTS, (there will be only one instance of STUDENTS because of the nature of description sets), and that the role name STUD# will serve as an identifier in the description set. The entry for STUDENTS.ADVISOR means that instances of this role name type will be part of the ESM and that in each instance the parameter Attribute Domain will have the value NAMES.

FIGURE 1-3. Decription Set Catalog

Element Name	Function	Identifiers	Attribute Domain
STUDENTS	DSN	(STUD#)	
STUDENTS.STUD#	RN		NUMBERS
STUDENTS.ADVISOR	RN		NAMES
SCHEDULE	DSN	(STUD#,COURSE)	
SCHEDULE.STUD#	RN		NUMBERS
SCHEDULE.COURSE	RN		COURSES
SCHEDULE.GRADE	RN		GRADES
COURSES	DSN	(COURSE)	
COURSES.COURSE	RN		COURSES
COURSES.INSTR	RN		NAMES
INSTRUCTORS	DSN	(INSTR)	
INSTRUCTORS.INSTR	RN		NAMES
INSTRUCTORS.DEGREE	RN		DEGREES

DSN=Description Set Name
RN=Role Name

1.2 - String Model

The String Model is the sub-model of the DIAM in which access paths are defined to connect Attribute Domain/Role Name/Attribute Value triplets and collections of these triplets. Since the thread of a connection has the effect of tying together triplets and collections, it is called a string, and hence the name String Model.

Strings are needed so that triplets in the interior of the model can be accessed as efficiently as required. Interior locations are those which are not listed as entry points to the data base, and to reach such locations it is necessary to follow a chain of address pointers. The pointers may be computed, explicitly stored, or implied as in a linear sequential search. Strings provide a means of conceptualizing a pointer structure, and the linking of two elements by a string means that once the first element has been retrieved a pointer to the second can somehow be obtained.

One of the main themes of the DIAM is modularity. An attempt has been made to construct a model from a concise set of component types, and as an instance of a given type of component occurs it is described by values of the parameters that are appropriate to its type. For example role names are described by the parameters Attribute Domain and Attribute Value. Whenever an instance of a given role name occurs it is defined by values for these parameters.

One of the advantages of parameterization is that if all the instances of a given type of component share the same values for one or more of their parameters, these values can be factored out and placed in a single type description for that component. The catalog is the listing of the type descriptions for all components. Parameter values that are given in the catalog need not be stored with each instance of the component. For example all instances of a given role name will utilize the same attribute domain, and hence the domain name can be factored into the catalog as was done in Figure 1-3. Attribute values, however, may vary with each role name instance and so must appear in the instance representation.

In other words, a role name is one of the basic components of the model. A specific role name, say STUD#, is a role name type for the model under consideration. Every triplet containing STUD# that appears in the model is an instance of the role name type STUD#. There will be a catalog entry for the role name STUD# that will give the associated attribute domain name, say NUMBERS, since all instances of the role name will use values from this same domain. The only part of a triplet that must appear as actual stored data is the attribute value; the domain name and role name are constant over all occurrences of this type of triplet and have been factored into the catalog. Similarly, named strings are basic components of the model.

The elements of the String Model will be discussed mainly in terms of their parameters. Since the purpose of a string is to form links among instances of role name triplets and/or other strings, the parameters appropriate to a string will be: 1) an Exit List naming the types of elements that the string connects, 2) an ON List naming the strings which have this string on their Exit lists, and 3) conditional information specifying what instances are to be linked. This information will be constant over all instances of a given string type and so can always be factored into the catalog.

There are three general kinds of strings, Attribute Strings (A-strings), Entity Strings (E-strings), and Link Strings (L-strings), which differ in the types of elements over which they can be defined and hence in the kind of collections which they can form. When a string type is created it is given a name that is unique across the String Model. For example A1,A2, and A3 might be the names of A-string types. There will be a catalog entry for each type and possibly many instances of each string throughout the model.

<u>A-strings</u> - The function of an A-string is to link role name triplets that are part of the same entity description. All the triplets in the description need not be on the same A-string, but they must be on some A-string if they are to be accessible. If an A-string is defined over a description set, there will be one instance of the A-string for each entity description in the set. A-string parameters are:

1) Exit List of role names that appear on instances of the A-string;
2) ON List of strings on which instances of this A-string appear.

As an example we could define an A-string, say SA1, over the description set STUDENTS. SA1 might have an exit list given by EXL=(STUD#,ADVISOR) indicating that there is an instance of SA1 for each student and that each instance connects two triplets, one containing the role name STUD# and the other containing ADVISOR. The ON list parameter cannot be specified until other strings have been defined for this A-string to be on.

<u>E-strings</u> - E-strings link homogeneous elements, that is they form collections which are subsets of the set of instances of one other string type. The collections are formed according to some criteria which is one of the E-string parameters.

Suppose A-string SA1 has been formed with an instance for each student. We might use an E-string, say SE1, to link all instances of SA1, thus giving access to a complete list of students. We might also define an E-string, SE2, which links the instances of SA1 in which the values associated with the role name ADVISOR are the same. This produces lists of students who have the same advisor. There will be one instance of SE2 for each unique value of advisor. E-string parametrs are:

1) Exit List, (will have only one element);

2) ON List;

3) Subset Selection Criterion (SSC) which may be:

a) A conditional statement specifying a boolean condition on attribute values associated with each instance of the string over which the E-string is defined. If the boolean condition is TRUE the instance is included on the E-string.

b) A conditional statement specifying a previously defined collection of which a string instance must be a part if it is to be included on the E-string.

c) A partition statement which names one of the role names on the strings over which the E-string is defined. All instances which have the same value for this role name will be on one E-string instance, those with another value on another instance, etc. In this way the set of instances over which the E-string is defined is partitioned according to values of the specified role name. The form of this parameter is SSC=: ROLENAME = Value(n), and instances of the E-string defined in this way are identified by subscripting the E-string name with a value of the partitioning role name.

4) Order On criterion (OO) is a parameter unique to E-strings. Since the elements on an E-string are all of the same type, it is meaningful to define an order over them. Stated in the form OO=ROLENAME, this parameter means that the string instances on each E-string instance are to be linked in sequential order by the values associated with ROLENAME.

The two E-strings, SE1 and SE2, defined above would have the catalog entries:

SE1 ESG EXL=(SA1); OO=(STUD#); ON=...

SE2 ESG EXL=(SA1); SSC=:(ADVISOR=Value(n)); OO=(STUD#); ON=...

(ESG stands for E-string)

<u>L-strings</u> - The function of an L-string is to link heterogeneous elements, that is instances of different previously defined strings. The exit list of the L-string will name a number of other strings and each instance of the L-string will contain one and only one instance of each named string. The instances are linked on the basis of a match of attribute values that appear on each. There will be an L-string instance for each possible value of the specified attribute.

L-string parameters are:

1) Exit List

2) ON List

3) Match Criteria (MC) which specifies the role names from each string on the exit list whose values must match to determine inclusion in a given L-string instance. This parameter appears in the catalog as

MC=(Description Set Name.Role Name = Description Set Name.Role Name = ...)

As an illustration of an L-string, suppose we have the ESM given in Figure 1-3 and that we wish to link each course name with the degree held by the instructor of that course. The following strings would be needed:

CA1 ASG EXL=(COURSE,INSTR); ON= CL1

IA1 ASG EXL=(INSTR,DEGREE); ON= CL1

CL1 LSG EXL=(CA1,IA1); MC=(COURSES.INSTR = INSTRUCTORS.INSTR); ON=...

CA1 and IA1 are A-strings. An instance of CA1 ties together the triplets that describe a course and an instance of IA1 ties together the triplets that describe an instructor. CL1 is an L-string and an instance of CL1 ties together an instance of CA1 and an instance of IA1 for which the values of their common role name, INSTR, are the same. In this case there will be an instance of CL1 for each instance of CA1, and in general there will be one instance of the L-string for each instance of the first string on its exit list.

A String Model is a network of E, L, and A-strings defined over the role name triplets of an Entity Set Model. Figure 1-4 gives a graphic representation of the sample strings defined in this section. An instance of a string with no incoming arrow such as SE1 is considered to be an entry point into the data base. It can be seen how data values could be retrieved by starting at an entry point and tracing through connected string instances to the role name triplets that are desired. It is in this sense that strings constitute access paths to the stored data.

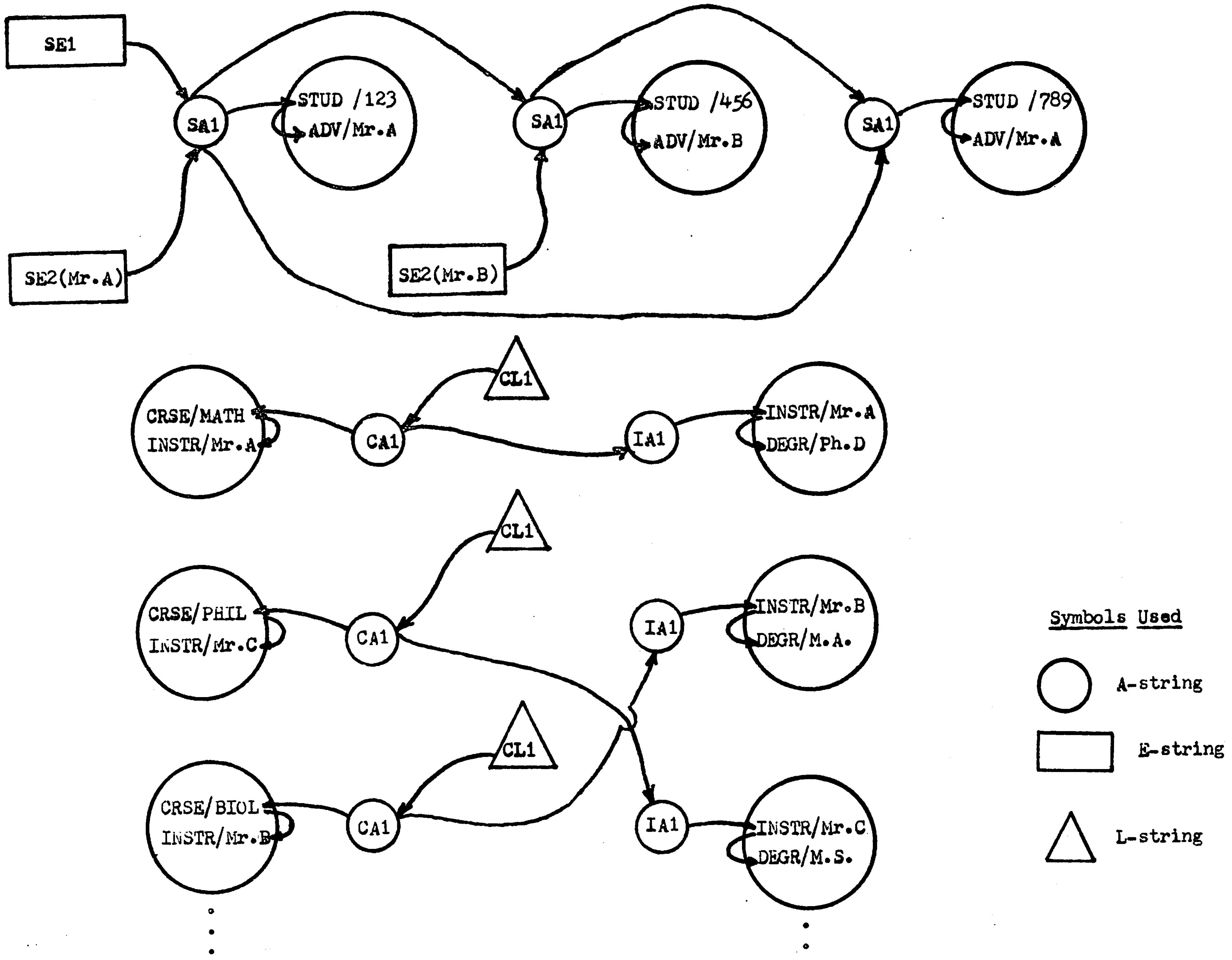

SE1
SA1
STUD /123
ADV/Mr.A
SA1
STUD /456
ADV/Mr.B
SA1
STUD /789
ADV/Mr.A
SE2(Mr.A)
SE2(Mr.B)
CL1
CRSE/MATH
INSTR/Mr.A
CA1
IA1
INSTR/Mr.A
DEGR/Ph.D
CL1
CRSE/PHIL
INSTR/Mr.C
CA1
IA1
INSTR/Mr.B
DEGR/M.A.
CL1
CRSE/BIOL
INSTR/Mr.B
CA1
IA1
INSTR/Mr.C
DEGR/M.S.
Symbols Used
A-string
E-string
L-string

FIGURE 1-4

1.3 - Encoding Model

At the encoding level of the DIAM we are concerned with representing the elements of the String Model in a form suitable for storage on a physical device. It is assumed that the creation of symbol groups to encode individual attribute values, string names and structural information is handled automatically. What is of interest here is the creation of virtual pointers to implement the links prescribed by the String Model and the placement of symbol groups which represent names, pointers, etc. in a data stream which will reside on some storage device and constitute the physical model of the real world information.

In order to be free from the constraints of any particular physical device, symbol groups are viewed as being placed in a conceptualized storage area called a Linear Address Space (LAS). An LAS consists of an array of addressable locations, each of which, for the purposes of this discussion, can hold the symbol group that encodes a single alphanumeric character. Several named LASs may be used to form the Encoding Model.

Information from the String Model will be coded in one of two general formats. First, actual attribute values will appear simply as groups of characters in contiguous LAS locations. Second, all other information will be organized in Basic Encoding Units (BEUs). Specifically, there will be a BEU for each string and role name, and string names, pointers, and other control information will be placed in the appointed locations in the BEU format. The format for a single BEU is as follows:

LABEL	APTR1	APTR2	...	...	APTRn	VPTR	TERM

where

LABEL is the name of the String Model element which the BEU is encoding.
APTR1...APTRn are n Association Pointers. There is one APTR for each string which the element being encoded is on, and the APTR holds the LAS address of the next element on the string to which it corresponds.
VPTR is the Value Pointer which holds the LAS address of the first element in the collection defined by this BEU. If the BEU represents a string the VPTR will point to the first element on the string. If the BEU represents a role name the VPTR will point to an attribute value.
TERM contains termination information indicating where the collection defined by this BEU ends.

The four fields described above are the parameters for the Encoding Model. Since there will be one BEU for each element instance in the String Model, there will be sets of BEUs corresponding to element types, and BEUs in each set will share some common parameters. Once again we can take advantage of these similarities to factor out the common information and place it in catalog entries for each BEU type. For example all the BEUs representing instances of a single string type will have the same value in the LABEL field. The label value could, in many cases, be placed in the catalog entry for the BEU type. A great deal of information can be removed from the data stream by this kind of factoring.

The catalog entries for BEU parameters will have the formats given below. In general the entry will either contain the parameter value or it will describe the field in the BEU that holds the parameter value. When the second method is used, the entry may give the size of the field and the units (bits, bytes,etc.) in which the size is measured or it may give a termination symbol that will appear in the BEU to denote the end of the field.

A typical catalog entry might have the form VALUE=q to indicate that this parameter will have the value q for all instances of the string type. In this case the parameter value has been completely factored into the catalog. If the parameter value is not factored, but appears in each BEU instance, the catalog entry must describe the field in which it is placed so that a decoding mechanism can locate it. This may be done by giving the size of the field and the units in which it is measured using the entries SIZE=x, UNITS=y . For example, if the parameter value is to appear in a field that is 4 bytes long the entry SIZE=4 , UNITS = bytes would be used. Rather than specifying a fixed field size, it might be convenient to use a termination marker to delimit the field. In this case the entry TERMINATOR=z is used. If, for instance, the end of a field is to be marked by the character # the entry would be TERMINATOR= #.

Using the above conventions, the parameters that describe a given BEU type will appear in the catalog as follows. (Underling indicates reserved words and { } denotes alternative forms.)

LABEL: If the string name is factored into the catalog, its value can be given by the entry STRING LABEL = q where q is the string name. If the name is not factored, the field in which it appears may be described by one of the following entries $\left\{\begin{array}{l}\text{SIZE=x, UNITS=y;}\\ \text{TERMINATOR=z;}\end{array}\right\}$

APTR: Since there will be an association pointer for each string (list) the BEU is ON, each APTR must be identified with the string to which it applies. The catalog entry to do this will be STRING = (string name) . The address that is the pointer will be given in terms of a displacement from an origin within a linear address space. The LAS must be identified either by placing the name in the catalog or by providing a field in the BEU. One of the following entries should be used

$$\left\{\begin{array}{l}\underline{\text{LAS}}\ \underline{\text{NAME}}\ \underline{\text{VALUE}} = q;\\ \underline{\text{LAS}}\ \underline{\text{NAME}}\ \underline{\text{SIZE}} = x,\ \underline{\text{UNITS}} = y;\\ \underline{\text{LAS}}\ \underline{\text{NAME}}\ \underline{\text{TERMINATOR}} = z;\end{array}\right\}$$

Similarly, the displacement value can be factored or not, and so the possible entries are

$$\left\{\begin{array}{l}\underline{\text{DISPLACEMENT}}\ \underline{\text{VALUE}} = q;\\ \underline{\text{DISPLACEMENT}}\ \underline{\text{SIZE}} = x,\ \underline{\text{UNITS}} = y;\\ \underline{\text{DISPLACEMENT}}\ \underline{\text{TERMINATOR}} = z;\end{array}\right\}$$

A default on the displacement entry will indicate a value of zero. The next entry will give the origin from which the displacement is to be measured. The form of the entry will be

$$\text{ORIGIN} = \left\{\begin{array}{l}\underline{\text{START}}\\ \underline{\text{NEXTI}}\\ \underline{\text{NEXTC}}\\ \underline{\text{AFTER}}\end{array}\right\}$$

If START is used then the displacement is to be measured from the beginning of the LAS that has been specified. NEXTI means that the origin is a point immediately following the present BEU. NEXTC designates as the origin a point immediately following the present BEU plus any contiguous portions of its defining collection. AFTER designates a point following the final portion of the defining collection regardless of whether the elements of the defining collection are contiguous or not. The meaning of these terms will be further illustrated in the example given below.

The final parameter in an APTR entry gives the units in which the displacement is to be measured. It has the form $\underline{\text{DISPLACEMENT}}\ \underline{\text{UNITS}} = y$.

VPTR: Since the VPTR is also a pointer it has the same parameters as the APTR except that the STRING entry is not needed because there can be only one VPTR in each BEU. In addition the displacement may be given by a function to support hash addressing mechanisms. In this case the displacement entry is DISPLACEMENT FUNCTION = function

TERM: A collection may be terminated in several ways. First, there may be a count of some kind of units such as bits, bytes or collection elements. The entry would be

$$\left\{\begin{array}{l} \underline{\text{SIZE}} = x,\ \underline{\text{UNITS}} = y_1, \\ \underline{\text{TERMINATOR}} = z, \\ \underline{\text{VALUE}} = q, \end{array}\right\} \qquad \underline{\text{COUNTUNITS}} = y_2$$

Second, a termination symbol may be used and it will appear in some specified field of the last BEU in the collection. The required entry is

$$\underline{\text{TERMINATOR}} = q,\ \underline{\text{FIELD}}\ \underline{\text{USED}} = \left\{\begin{array}{l} \text{LABEL} \\ \text{APTR} \\ \text{VPTR} \\ \text{TERM} \end{array}\right\} ;$$

Lastly, a given termination address may be specified using entries of the same form as for a VPTR displacement.

To illustrate the specification of an Encoding Model, consider a string structure described by the following string catalog entries:

String Name	Type	Parameters
A1	ASG	EXL=(R1); ON=(L1)
A2	ASG	EXL=(R2,R3); ON=(E2,L1)
E1	ESG	EXL=(L1); OO=R1; ON=(ENTRY)
E2	ESG	EXL=(A2); OO=R2; ON=(ENTRY)
L1	LSG	EXL=(A1,A2); MC=(A1.R1=A2.R2); ON=(E1)
R1	RN	ON=(A1)
R2	RN	ON=(A2)
R3	RN	ON=(A2)

(See Figure 1-5)

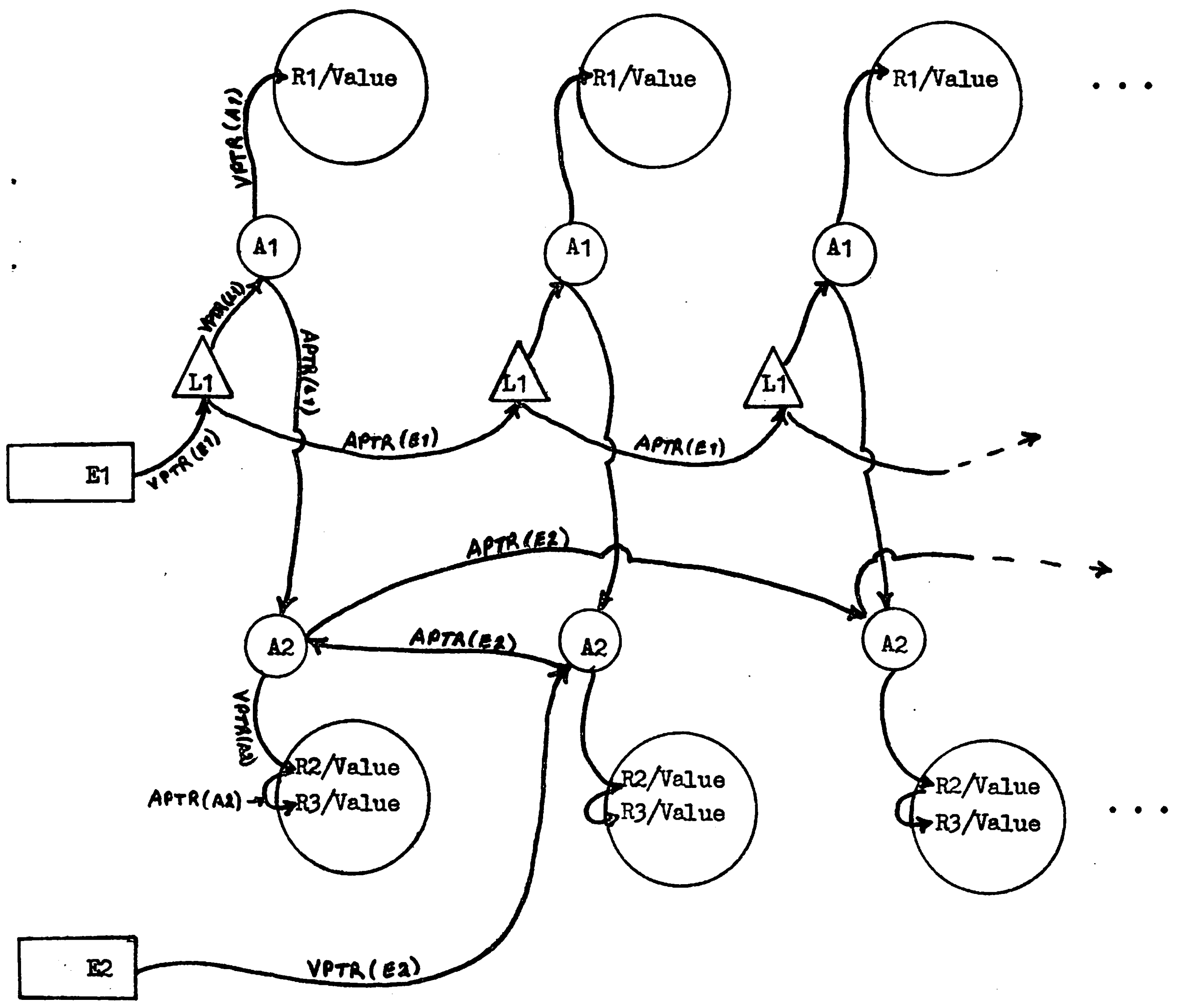


FIGURE 1-5: Graphic representation of string structure given on page 24.

Symbols Used

○	A-string (ASG)
▭	E-string (ESG)
△	L-string (LSG)

One possible encoding of this structure would be implemented by the following encoding level catalog entries.

Entry for String A1:

```
LABEL: STRING LABEL=A1
APTR:  STRING=L1;LAS NAME VALUE=S2;DISPLACEMENT SIZE=4,UNITS=Bytes;
       ORIGIN=START;DISPLACEMENT UNITS=Bytes
VPTR:  LAS NAME VALUE=S1; ORIGIN=NEXTI
TERM:  TERM(R1)
```

This entry means that the string A1 has its LABEL factored into the catalog. It has one APTR for the string L1. The pointer value is not factored, but will be found in each instance of an A1 BEU in a 4 byte field. The value in this field is the displacement from the START of LAS S2 to the BEU for the next element on L1. The VPTR points to the first element on A1's Exit List, namely the role name R1. The displacement is zero and the ORIGIN=NEXTI, meaning that the BEU for R1 is located in LAS S1 immediately following the BEU for A1. The end of A1's defining collection coincides with that of R1.

Entry for String A2:

```
LABEL: STRING LABEL=A2
APTR:  STRING=E2;LAS NAME VALUE=S2;DISPLACEMENT SIZE=4,UNITS=Bytes;
       ORIGIN=START;DISPLACEMENT UNITS=Bytes
APTR:  STRING=L1;DISPLACEMENT VALUE=Ø
VPTR:  ORIGIN=NEXTI
TERM:  TERM(R3)
```

Each instance of A2 is ON two strings (E2,L1) and so has two APTRs. The Aptr for E2 appears in a 4 byte field in the BEU, while the APTR for L1 has a null value (Ø) since A2 is the last element on the defining collection of L1.

Entry for String L1:

```
LABEL: STRING LABEL=L1
APTR:  STRING=E1;LAS NAME VALUE=S1;ORIGIN=NEXTC
VPTR:  ORIGIN=NEXTI
TERM:  TERM(A2)
```

Each instance of L1 begins a collection of A-string instances (A1,A2). A1 is located immediately following L1, but the matching instance of A2 is in a different LAS. The use of NEXTC in the APTR of an instance of L1 means that the next element in the defining collection of E1 (that is the next instance of L1 on the E-string E1) is located after the contiguous portions of the defining collection of L1, that is after the instance of A1. The terminator of an instance of L1 is defined to be the terminator of A2, since that is the last element on L1.

Entry for String E1:

```
LABEL: STRING LABEL=E1
VPTR: LAS NAME VALUE=S1;ORIGIN=START
TERM: VALUE=68;COUNTUNITS=L1 instances
```

An E1 BEU has no APTR because it is an entry point and not on any higher string. The first instance in the defining collection of E1 will be found at the start of LAS S1 and the collection will terminate after 68 instances of L1.

Entry for E2:

```
LABEL:  STRING LABEL= E2
VPTR:  LAS NAME VALUE= S2;  DISPLACEMENT SIZE= 4, UNITS= Bytes;
   ORIGIN=START;  DISPLACEMENT UNITS= Bytes.
TERM:  VALUE= 'LAST';  FIELD USED= APTR.
```

The collection of A2 instances that defines E2 is terminated by the character string 'LAST' in the final element in the collection.

Entry for R1:

```
LABEL: STRING LABEL= R1
APTR: STRING= A1; VALUE= Ø
VPTR: ORIGIN=NEXTI
TERM: VALUE= 16; COUNTUNITS= Bytes
```

The VPTR of NEXTI means that the value associated with R1 will be located immediately following the BEU for R1. The defining collection of R1 (i.e., the associated value field) will always be 16 bytes long.

Entry for R2:

```
LABEL:  STRING LABEL=R2
APTR:  STRING= A2; ORIGIN= AFTER
VPTR: ORIGIN= NEXTI
TERM: VALUE= 16; COUNTUNITS= Bytes.
```

The APTR points to the next element on the A2 string, namely an instance of R3. AFTER means that this instance is located after the value field associated with R2.

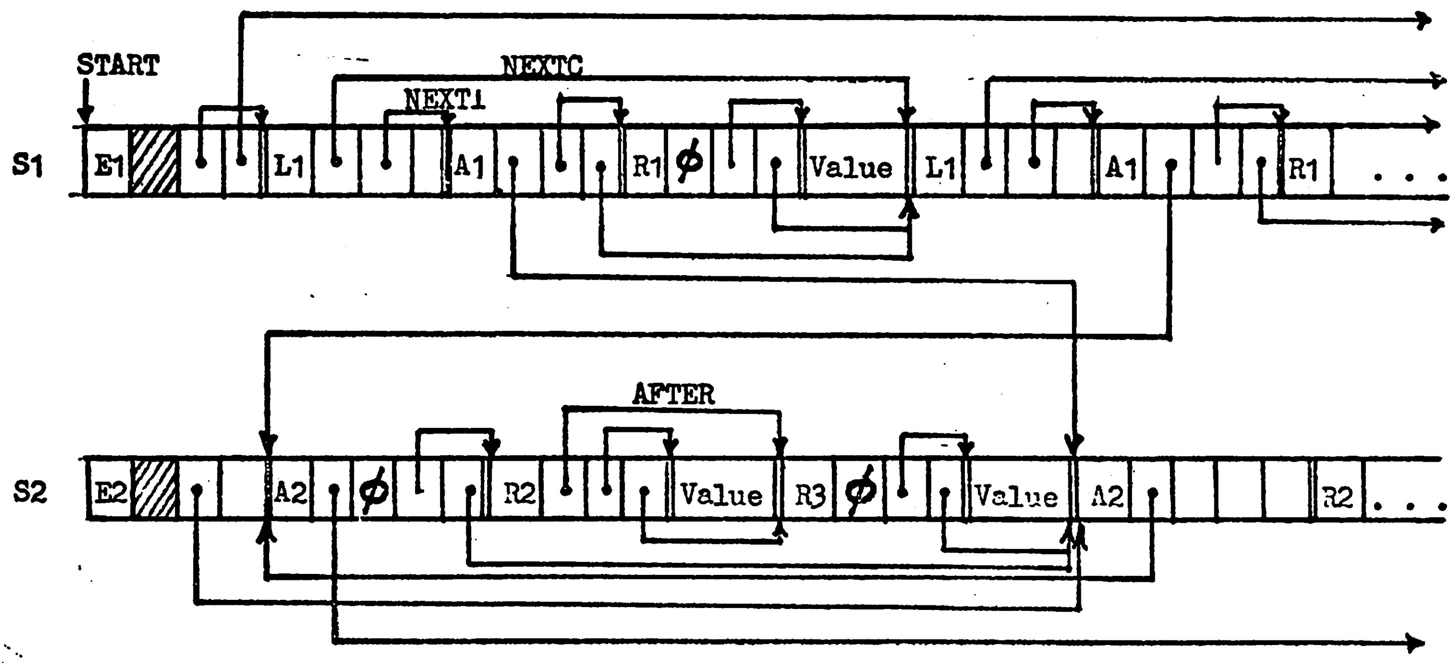
Entry for R3:

> LABEL: <u>STRING LABEL</u>= R3
> APTR: <u>STRING</u>= A2; <u>VALUE</u>= ∅
> VPTR: <u>ORIGIN</u>= <u>NEXTI</u>
> TERM: <u>VALUE</u>= 12; <u>COUNTUNITS</u>= Bytes

The following diagram is a conceptual representation of the two LASs described above. BEUs are separated by double lines and the format of a BEU is

LABEL	APTR	VPTR	TERM

. Notice the Value fields which hold the actual data values interspersed with the BEUs. The Value fields are pointed to by role name BEUs.



Since in reality much of the pointer information shown in the above diagram is factored into the catalog, the actual data stream would appear as

	A1		A1		A1	
S1	APTR(L1)	VALUE(R1)	APTR(L1)	VALUE(R1)	APTR(L1)	. . .
	4 Bytes	16 Bytes				

	E2	A2			A2		
S2	VPTR(E2)	APTR(E2)	VALUE(R2)	VALUE(R3)	APTR(E2)	VALUE(R2)	. . .
	4 Bytes	4 Bytes	16 Bytes	12 Bytes			

1.4 - Physical Device Model

At the physical device level we are concerned with the way in which coded symbol groups are entered on physical storage media. This involves the assignment of BEUs from the Encoding Model to storage locations on a physical device such as a magnetic drum or disk. BEUs will not be assigned individually, but rather in bunches called Contiguous Data Groups (CDGs). A CDG is defined as the largest possible set of BEUs and attribute values which are associated by contiguity within an LAS and which encompass at most one instance of the highest level E-string in the LAS.* In general the assignment procedure is: Associate an LAS with a specific portion of physical storage and then define rules for the placement of CDGs from the LAS within the physical space.

Physical Sub-divisions (PHYSD) are named address areas on a storage device. (e.g. disk pack, cylinder, track, block, etc.) As with other elements of the DIAM, a PHYSD name identifies a sub-division type of which there may be many instances. Suppose a type of sub-division named BLKA is specified as an area of 100 contiguous bytes, and BLKB is given 25 bytes. Then a sub-division named TRKA might consist of one instance of BLKB followed by as many instances of BLKA as can fit on a track. In this way PHYSDs can be built up from the smallest units of addressable storage (bytes) and each instance of a PHYSD type will have a uniform composition.

* Collections of BEUs that encode string structures are in fact multilist data structures in which each BEU contains pointers to the next items in the lists of which it is a member. In many cases these pointers are factored from the data stream and, during decoding, their values must be implied from the relative positions of BEUs in the linear address space. When pointers are stored in this fashion, the contiguity of the BEUs involved must be maintained in the physical model. Contiguous Data Groups are sets of BEUs that must be stored contiguously to preserve implied pointer values.

The parameter values that describe a PHYSD type may be completely factored into the catalog since they will be the same for all instances. The catalog entries will be:

NAME	Name assigned to the PHYSD
FORMAT	This is basically a list of PHYSDs which compose this PHYSD. Before each PHYSD name in the list is the number of repetitions of that PHYSD that will appear. If the number of repetitions is given as X, then there will be as many instances of the PHYSD as are required to fill the PHYSD being described. After each PHYSD name in the list will be an indication of where that PHYSD will start within the scope of an instance of the PHYSD being described. If <u>AFTER</u> is used the first instance will be placed in the next available appropriate sub-division. Finally, there will be an indication of whether the PHYSD is to be filled by normal loading procedures, (indicated by <u>FILL</u>), or reserved for special use,(indicated by <u>RESERVE</u>).
CONTROL AREA	The names of PHYSDs reserved for control information.
SPAN	A value of YES for this parameter means that if a CDG will not fit completely into this PHYSD the first part will be placed in this sub-division and the remainder will span its boundary and fall into the next appropriate sub-division. A value of NO means that the entire CDG is to be placed in the next subdivision leaving the remaining space in this sub-division empty.
TYPE	The portion of a physical device for which this PHYSD will serve as a template. It is assumed that the storage device being used is divided into predefined areas with system recognizable boundaries, (e.g. cylinders and tracks on a disk). One of these areas, say TRACK,

will be the value of the parameter TYPE, indicating that each instance of this PHYSD will occupy one track on a disk, and that the catalog entry for this PHYSD will provide a template for the storage locations between the boundaries of the track.

Some examples of catalog entries for PHYSDs are:

```
NAME = TRKA     FORMAT = (1(BLKB),START = BLOCK = 0,FILL)
                         (X(BLKA),START = BLOCK = AFTER, FILL)
                CONTROL AREA = BLKB
                TYPE = TRACK
```

This entry describes a physical sub-division called TRKA. From the value of the TYPE parameter we know that each instance of TRKA will take up one track on a disk. The FORMAT parameter shows that each instance of TRKA will be divided into 1 instance of a sub-division called BLKB followed by as many instances of a sub-division called BLKA as can fit on the track. The entire track is to be filled at load time, and BLKB will contain control information. The exact nature of this control information will not be specified here. The next two entries give the composition of the PHYSDs BLKA and BLKB, which are merely blocks of 100 and 25 contiguous bytes, respectively. Since a byte is the smallest unit of addressable storage, no smaller sub-divisions need be defined.

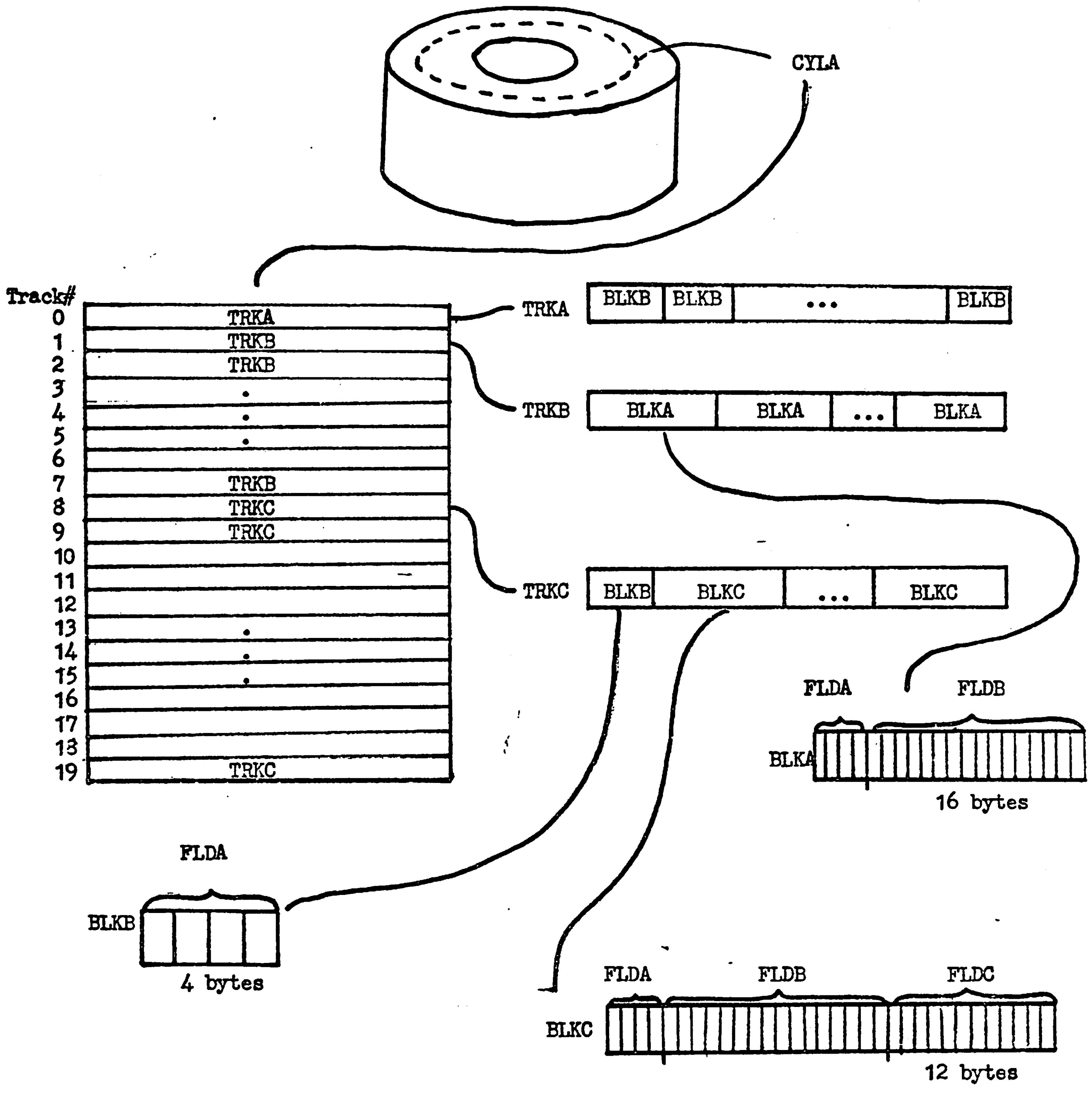
```
NAME = BLKA     FORMAT = (100(BYTES),START = BYTE = 0, FILL)
                TYPE = BLOCK

NAME = BLK      FORMAT = (25(BYTES),START = BYTE = AFTER, FILL)
                TYPE = BLOCK
```

```
TRKA    FORMAT = (X(BLKB), START = BLOCK = 0, RESERVE)
        SPAN = NO
        TYPE = TRACK
TRKB    FORMAT = (x(BLKA), START = BLOCK = 0, FILL)
        SPAN = NO
        TYPE = TRACK
TRKC    FORMAT = (1(BLKB), START = BYTE = 0, FILL)
                 (X(BLKC), START = BYTE = 4, FILL)
        SPAN = YES
        TYPE = TRACK
BLKA    FORMAT = (1(FLDA), START = BYTE = 0, FILL)
                 (1(FLDB), START = BYTE = 4, FILL)
        TYPE = BLOCK
BLKB    FORMAT = (1(FLDA), START = BYTE = 0, FILL)
        TYPE = BLOCK
BLKC    FORMAT = (1(FLDA), START = BYTE = 0, FILL)
                 (1(FLDB), START = BYTE = 4, FILL)
                 (1(FLDC), START = BYTE = 20, FILL)
        TYPE = BLOCK
FLDA    FORMAT = (4(BYTES), START = BYTE = 0, FILL)
        TYPE = FIELD
FLDB    FORMAT = (16(BYTES), START = BYTE = 0, FILL)
        TYPE = FIELD
FLDC    FORMAT = (12(BYTES), START = BYTE = 0, FILL)
                 TYPE = FIELD
```

Figure 1-6 shows how these PHYSDs would be arranged on a disk.

FIGURE 1-6


CYLA
Track#
0 TRKA
1 TRKB
2 TRKB
3
4
5
6
7 TRKB
8 TRKC
9 TRKC
10
11
12
13
14
15
16
17
18
19 TRKC
TRKA
BLKB BLKB ... BLKB
TRKB
BLKA BLKA ... BLKA
TRKC
BLKB BLKC ... BLKC
FLDA FLDB
BLKA
16 bytes
FLDA
BLKB
4 bytes
FLDA FLDB FLDC
BLKC
12 bytes

Once the organization of physical storage has been specified, the LASs from the Encoding Model can be assigned to particular PHYSDs. This mapping is accomplished by a catalog entry for each LAS containing the LAS name followed by a nested expression in which the PHYSDs that will hold the LAS are named. Within this expression each PHYSD name may be followed by the names of its component PHYSDs. Only those components will be listed that are to be used in storing the LAS.

For example, suppose a disk pack PACKA is divided into 5 cylinders of type CYLA and 195 cylinders of type CYLB. Each CYLB is in turn divided into 2 tracks of type TRKA and 18 tracks of type TRKB. Then to store LAS A only on TRKB type tracks in CYLB cylinders the catalog entry for the LAS would be

LAS NAME = A (PHYSD = PACKA (PHYSD = CYLB (PHYSD = TRKB)))

Since CYLA and TRKA are not mentioned, PHYSDs of these types are not used in storing this LAS. Because no sub-divisions of TRKB are listed it is assumed that the entire track is to be used.

The particular instances of a PHYSD type that are to be used to store an LAS can be specified by following the PHYSD name with a pair of values (START,END) where START is the number of the first instance of this PHYSD type that will be used to store the LAS. END is the number of the last instance used. For instance the entry

LAS NAME = A (PHYSD = PACKA (PHYSD = CYLB (PHYSD = TRKB(4,10))))

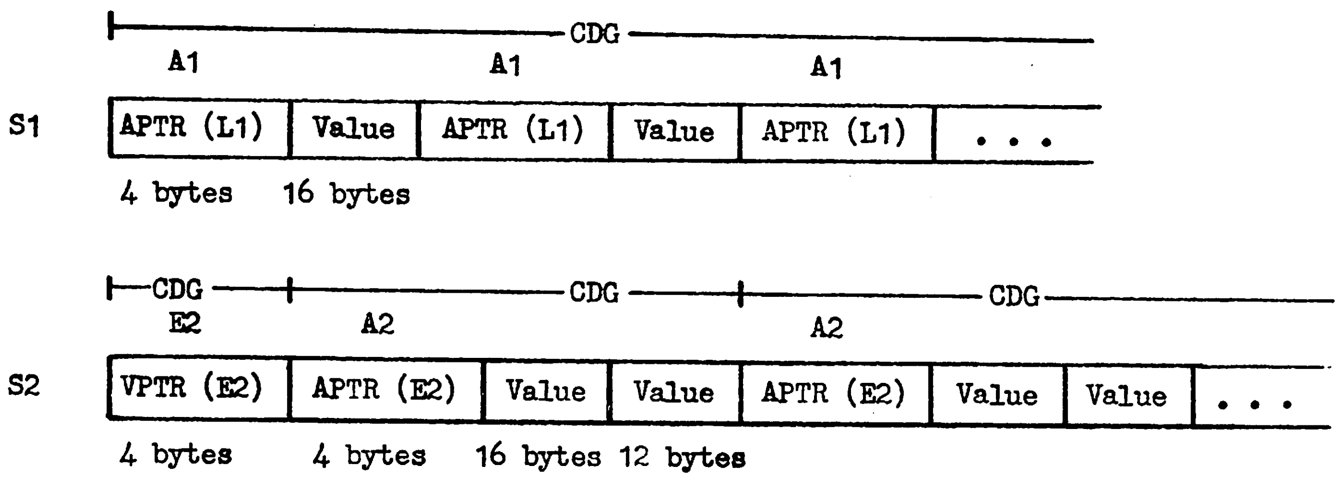
would mean that the LAS A is to be mapped into each cylinder of PACKA starting at the 4th track of type TRKB and extending through the 10th track of type TRKB.

As a further example, consider the LAS structure shown in Figure 1-5. In terms of string names these LASs are characterized by the sequences:

S1: E1 L1 A1 R1 Value L1 A1 R1 Value L1 A1 R1 Value ...

S2: E2 A2 R2 Value R3 Value A2 R2 Value R3 Value A2 R2 Value ...

The BEU data streams for S1 and S2 have the structures



S1 consists of only one contiguous data group since it contains one instance of an E-string whose ordering is implemented by contiguity. In S2 the components of each A-string A2 are related by contiguity but the A-string instances in E2 are not,(that is the APTR in A2 for E2 is not factored so physical contiguity need not be maintained). Thus we arê free to segment the BEU stream as we desire and can choose the CDG structure shown, perhaps in anticipation of future insertions. Assume that the physical storage device to be used is a magnetic disk and that 1 cylinder is available for the storage of LASs S1 and S2. We choose a physical organization described by the following catalog entries:

```
CYLA    FORMAT = (1 (TRKA), START = TRACK = 0, RESERVE)
                 (7 (TRKB), START = TRACK = AFTER, FILL)
                 (12(TRKC), START = TRACK = AFTER, FILL)
        CONTROL AREA = TRKA
        SPAN = NO
        TYPE = CYLINDER
```

The mapping of LASs S1 and S2 into the sub-divisions of the cylinder CYLA can be described by the following catalog entries:

```
    LAS NAME = S1    (PHYSD = CYLA (PHYSD = TRKB(1,5)))
```

which means that S1 is to be placed in CYLA starting with the 1st track of type TRKB and extending through the 5th track of type TRKB. All sub-divisions of these tracks will be spanned.

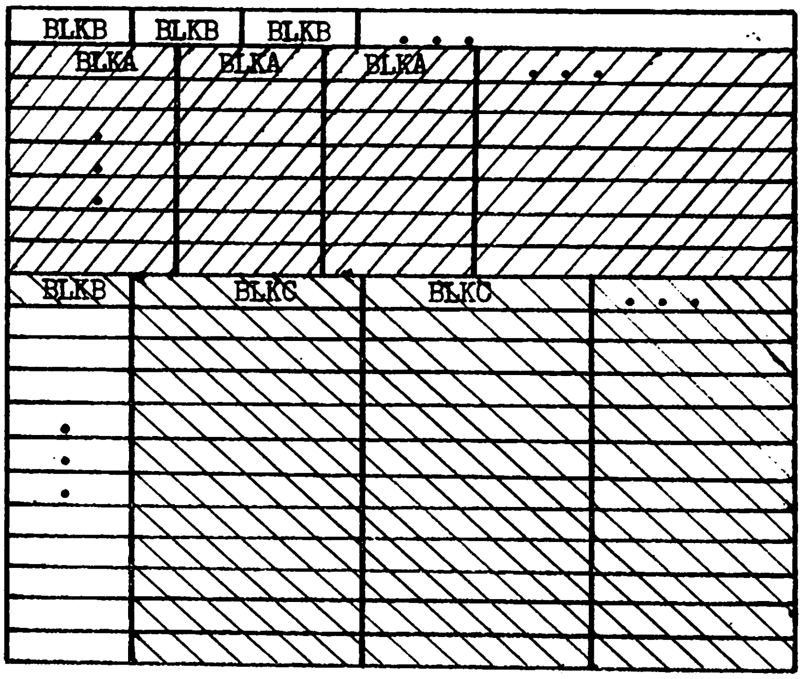
```
    LAS NAME = S2    (PHYSD = CYLA (PHYSD = TRKC(1), PHYSD = TRKC(2,12)
                               (PHYSD = BLKC)))
```

which means that S2 is to be placed in CYLA in tracks of type TRKC. In the 1st TRKC all sub-divisions (BLKB, BLKC) are to be used. In the 2nd - 12th TRKCs only blocks of type BLKC are to be used. The composition of CYLA after the placement of the LASs is shown in Figure 1-7.

The methods for constructing rules for the placement of Contiguous Data Groups within the physical sub-divisions assigned to an LAS are not well developed at this time. The simplest process is to take each CDG as it appears sequentially in the LAS and place it in the next available PHYSD that has been assigned to that LAS. This process is trivial if the physical organization has been designed to accomodate a known and fixed LAS structure; that is if PHYSDs have been tailored to hold the anticipated CDGs. This has been done in the above example, as fields have been provided to exactly fit the known size of BEU and value fields and blocks have been specifically designed to hold CDGs. The CDG placement procedure for this case can then be described as: Maintain for each LAS a pointer containing the address of the next unfilled block assigned to the LAS. To load an LAS begin at its START location and place each successive CDG on the physical device at the location given by the appropriate pointer. After each placement modify each pointer according to the catalog entry for the LAS.

FIGURE 1-7

Track#	Type	# Within Type
0	TRKA	1
1	TRKB	1
2	"	2
3	"	3
4	"	4
5	"	5
6	"	6
7	TRKB	7
8	TRKC	1
9	"	2
10	"	3
11	"	4
12	"	5
13	"	6
14	"	7
15	"	8
16	"	9
17	"	10
18	"	11
19	TRKC	12



▨ Space to be filled by CDGs from LAS S1

▧ Space to be filled by CDGs from LAS S2

2 - Example

This section traces the development of a DIAM for a specific set of data through the entity set level, string level, and encoding level. The structures presented here will be used to illustrate future points and discussions.

The data base being modelled contains the following information about employees:

Employee Number (EMP#)	Name (NAME)	Department (DEP)	Manager (MGR)	Project (PROJ)	Hours Worked (HRS)	Location (LOC)
1	AA	1	XX	8	14	A
				9	16	B
2	BB	2	YY	7	11	A
				9	14	B
3	CC	1	XX	8	20	A

Treating each column as an entity set, we notice that the following relations between sets hold.

EMP#	-	NAME	one-to-one	(1)
EMP#	-	DEP	many-to-one	(2)
EMP#	-	PROJ	many-to-many	(3)
PROJ	-	LOC	many-to-one	(4)
DEP	-	MGR	one-to-one	(5)
EMP#,PROJ	-	HRS	many-to-one	(6)

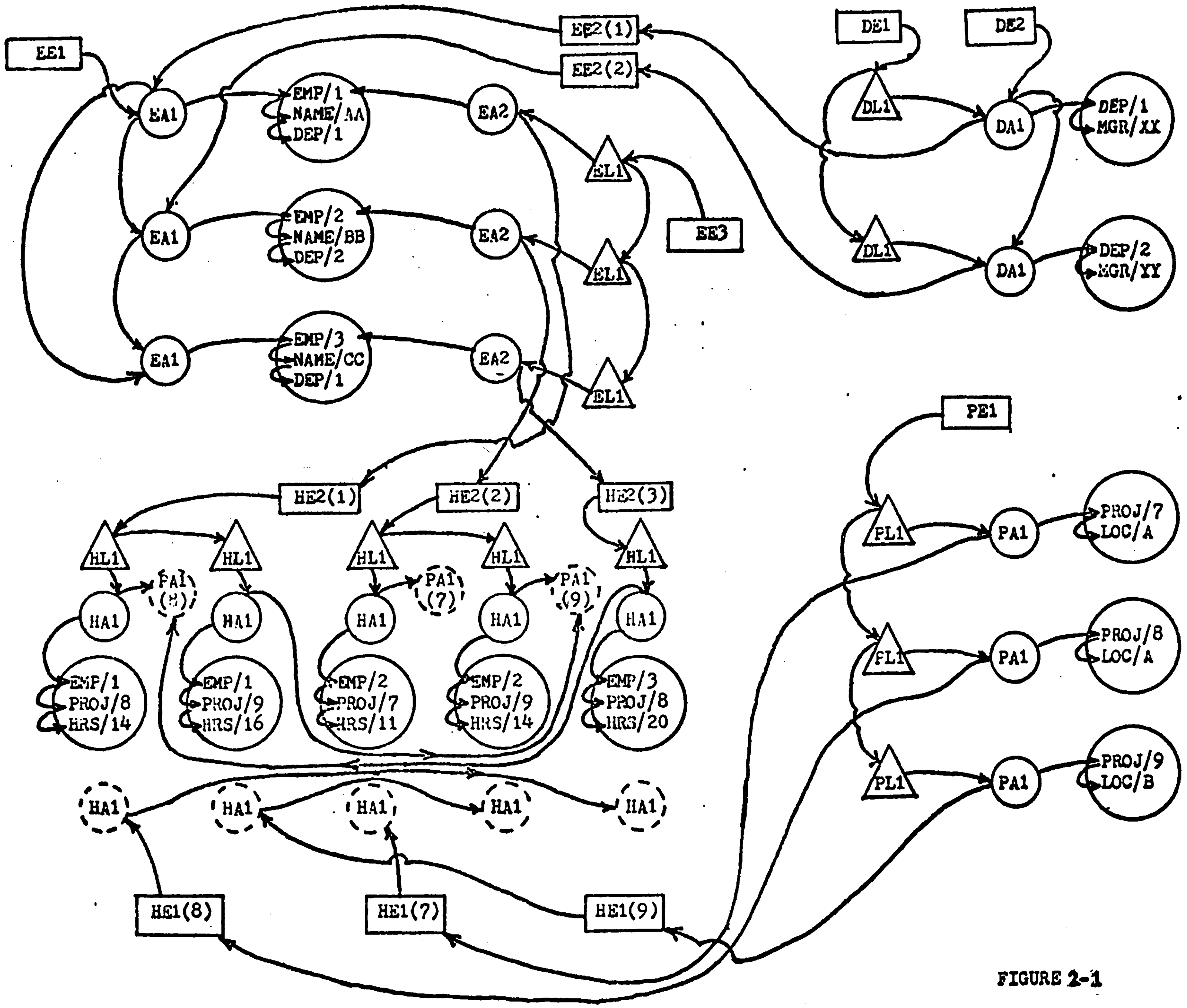
We now form the entity description sets by following the RNIA rules. Relations (1) and (2) indicate that NAME and DEP should be placed in a description set with EMP# as the identifier. Call this set EMP. Relations (3) and (6) dictate a description set with EMP#,PROJ combinations as the identifier and HRS as an associated role name. Call the set HOURS. From (4) we form a description set PROJT with role names PROJ and LOC. From (5) we place DEP and MGR in a set called DEPT with DEP as the identifier.

The Description Set Catalog resulting from these allocations is

Element Name	Function	Identifiers	Attribute Domain
EMP	DSN	(EMP#)	
EMP.EMP#	RN		NUMBERS
EMP.NAME	RN		NAMES
EMP.DEP	RN		NUMBERS
HOURS	DSN	(EMP#,PROJ)	
HOURS.EMP#	RN		NUMBERS
HOURS.PROJ	RN		NUMBERS
HOURS.HRS	RN		NUMBERS
PROJT	DSN	(PROJ)	
PROJT.PROJ	RN		NUMBERS
PROJT.LOC	RN		LETTERS
DEPT	DSN	(DEP)	
DEPT.DEP	RN		NUMBERS
DEPT.MGR	RN		NAMES

The string structures shown in Figure2-1 are chosen more or less arbitrarily and they are specified by the following String Catalog.

Name	Function	String Name	TYPE	Parameters
EMP	DSN			
EMP.EMP#	RN			ON=EA1,EA2)
EMP.NAME	RN			ON=EA1)
EMP.DEP	RN			ON=EA1)
		EA1	ASG	EXL=(EMP#,NAME,DEP);ON=EE1,EE2(n))
		EA2	ASG	EXL=(EMP#);ON=(EL1)
		EE1	ESG	EXL=(EA1);OO=(EMP.EMP#);ON=(ENTRY)
		EE2(n)	ESG	EXL=(EA1);SSC=:Value(n)=EMP.DEP ; OO=(EMP.EMP#);ON=(DL1)
		EL1	LSG	EXL=(EA2,HE2(p));MC=(EMP.EMP#= HOURS.EMP#);ON=(EE3)
		EE3	ESG	EXL=(EL1);OO=(EMP.EMP#);ON=(ENTRY)



EE1
EE2(1)
EE2(2)
EE3
DE1
DE2
EA1
EA2
EL1
DL1
DA1
EMP/1 NAME/AA DEP/1
EMP/2 NAME/BB DEP/2
EMP/3 NAME/CC DEP/1
DEP/1 MGR/XX
DEP/2 MGR/YY
PE1
PL1
PA1
PROJ/7 LOC/A
PROJ/8 LOC/A
PROJ/9 LOC/B
HE2(1)
HE2(2)
HE2(3)
HL1
HA1
PA1 (8)
PA1 (7)
PA1 (9)
EMP/1 PROJ/8 HRS/14
EMP/1 PROJ/9 HRS/16
EMP/2 PROJ/7 HRS/11
EMP/2 PROJ/9 HRS/14
EMP/3 PROJ/8 HRS/20
HE1(8)
HE1(7)
HE1(9)

FIGURE 2-1

Name	Function	String Name	Type	Parameters
HOURS	DSN			
HOURS.EMP#	RN			ON=(HA1)
HOURS.PROJ	RN			ON=(HA1)
HOURS.HRS	RN			ON=(HA1)
		HA1	ASG	EXL=(EMP#,PROJ,HRS);ON=(HE1(n),HL1)
		HE1(n)	ESG	EXL=(HA1);SSC=(:Value(n)=HOURS.PROJ); OO=(HOURS.EMP#);ON=(PL1)
		HE2(p)	ESG	EXL=(HL1);SSC=(:Value(p)=HOURS.EMP#); OO=(HOURS.PROJ);ON=(EL1)
		HL1	LSG	EXL=(HA1,PA1);MC=(HOURS.PROJ=PROJT.PROJ); ON=(HE2)
PROJT	DSN			
PROJT.PROJ	RN			ON=(PA1)
PROJT.LOC	RN			ON=(PA1)
		PA1	ASG	EXL=(PROJ,LOC);ON=(PL1,HL1)
		PL1	LSG	EXL=(PA1,HE1(n));MC=(PROJT.PROJ= HOURS.PROJ);ON=(PE1)
		PE1	ESG	EXL=(PL1);OO=(PROJT.PROJ);ON=(ENTRY)
DEPT	DSN			
DEPT.DEP	RN			ON=(DA1)
DEPT.MGR	RN			ON=(DA1)
		DA1	ASG	EXL=(DEP#,MGR);ON=(DL1,DE2)
		DL1	LSG	EXL=(DA1,EE2(n));MC=(DEPT.DEP=EMP.DEP#); ON=(DE1)
		DE1	ESG	EXL=(DL1);OO=(DEPT.DEP);ON=(ENTRY)
		DE2	ESG	EXL=(DA1);OO=(DEPT.DEP);ON=(ENTRY)

The next step is to specifiy the encoding of the strings and data values in Linear Address Spaces. We will use 4 LASs, one for each entity description set, and name them EMP,HOURS,PROJT,DEPT respectively. Each LAS is byte addressable with an initial address of 00. The Encoding Model is described by the following catalog entries.

String (or Role Name)	Type	Encoding Parameters
EA1	ASG	LABEL: STRING LABEL=(EA1)
		APTR: STRING=EE1; ORIGIN=AFTER
		APTR: STRING=EE2; DISPLACEMENT SIZE=3,UNITS=Bytes; ORIGIN=START;DISPLACEMENT UNITS =Bytes
		VPTR: ORIGIN=NEXTI
		TERM:TERM(EMP.DEP)
EE1	ESG	LABEL: STRING LABEL=EE1
		VPTR: ORIGIN=NEXTI
		TERM: VALUE=3, COUNTUNITS=EA1 instances
EA2	ASG	LABEL: STRING LABEL=EA2
		APTR: STRING=EL1;LAS NAME VALUE=HOURS;DISPLACEMENT SIZE=3, UNITS=Bytes;ORIGIN=START;DISPLACEMENT UNITS=Bytes
		VPTR:ORIGIN=NEXTI
		TERM: TERM(EMP.EMP#)
EE2(n)	ESG	LABEL: STRING LABEL=EE2
		APTR: STRING=DL1;DISPLACEMENT SIZE= ∅
		VPTR: DISPLACEMENT SIZE=2,UNITS=Bytes;ORIGIN=START
		TERM: TERMINATOR='EOB';FIELD USED=APTR
EL1	LSG	LABEL: STRING LABEL=EL1
		APTR: STRING=EE3;DISPLACEMENT VALUE=4;ORIGIN=NEXTI
		VPTR: ORIGIN=NEXTI
		TERM: TERM(HE2)
EE3	ESG	LABEL: STRING LABEL=EE3
		VPTR: ORIGIN=NEXTI
		TERM: VALUE=3,COUNTUNITS=EL1 instances
EMP.EMP#	RN	LABEL: STRING LABEL=EMP.EMP#
		APTR: STRING=EA1;ORIGIN=AFTER
		APTR: STRING=EA2;SIZE= ∅
		VPTR: ORIGIN=NEXTI
		TERM: VALUE=1,COUNTUNITS=Bytes

String	Type	Parameters
EMP.NAME	RN	LABEL: STRING LABEL=EMP.NAME
		APTR: STRING=EA1;ORIGIN=AFTER
		VPTR: ORIGIN=NEXTI
		TERM: VALUE=2, COUNTUNITS=Bytes
EMP.DEP	RN	LABEL: STRING LABEL=EMP.DEP
		APTR: STRING=EA1;DISPLACEMENT SIZE=ø
		VPTR: ORIGIN=NEXTI
		TERM: VALUE=1,COUNTUNITS=Bytes
HOURS.EMP#	RN	LABEL: STRING LABEL=HOURS.EMP#
		APTR: STRING=HA1;ORIGIN=AFTER
		VPTR: ORIGIN=NEXTI
		TERM: VALUE=1,COUNTUNITS=Bytes
HOURS.PROJ	RN	LABEL: STRING LABEL=HOURS.PROJ
		APTR: STRING=HA1;ORIGIN=AFTER
		VPTR: ORIGIN=NEXTI
		TERM: VALUE=1,COUNTUNITS=Bytes
HOURS.HRS	RN	LABEL: STRING LABEL=HOURS.HRS
		APTR: STRING=HA1;DISPLACEMENT SIZE=ø
		VPTR: ORIGIN=NEXTI
		TERM: VALUE=2,COUNTUNITS=Bytes
HA1	ASG	LABEL: STRING LABEL=HA1
		APTR: STRING=HE1; ORIGIN=AFTER
		APTR: STRING=HL1;IAS NAME VALUE=PROJT;DISPLACEMENT SIZE=2,
		UNITS=Bytes;ORIGIN=START;DISPLACEMENT UNITS=Bytes
		VPTR: ORIGIN=NEXTI
		TERM: TERM(HOURS.HRS)
HE1(n)	ESG	LABEL: STRING LABEL=HE1
		APTR: STRING=PL1;ORIGIN=AFTER
		VPTR: ORIGIN=NEXTI
		TERM: TERMINATOR='EP';FIELD USED=VALUE
HE2(p)	ESG	LABEL: STRING LABEL=HE2
		APTR: STRING=EL1;DISPLACEMENT SIZE=ø
		VPTR: ORIGIN=NEXTI
		TERM: TERMINATOR='EB';FIELD USED=VPTR
HL1	LSG	LABEL: STRING LABEL=HL1
		APTR: STRING=HE2;ORIGIN=AFTER
		VPTR: DISPLACEMENT SIZE=2,UNITS=Bytes;ORIGIN=START;
		DISPLACEMENT UNITS=Bytes
		TERM: TERMINATOR='LST';FIELD USED=APTR

String	Type	Parameters
PROJT.PROJ	RN	LABEL: STRING LABEL=PROJT.PROJ APTR: STRING=PA1;ORIGIN=AFTER VPTR: ORIGIN=NEXTI TERM: VALUE=1;COUNTUNITS=Bytes
PROJT.LOC	RN	LABEL: STRING LABEL=PROJT.LOC APTR: STRING=PA1;DISPLACEMENT SIZE=Ø VPTR: ORIGIN=NEXTI TERM: VALUE=1,COUNTUNITS=Bytes
PA1	ASG	LABEL: STRING LABEL=PA1 APTR: STRING=PL1;LAS NAME VALUE=HOURS;DISPLACEMENT SIZE=2, UNITS=Bytes;ORIGIN=START;DISPLACEMENT UNITS=Bytes APTR: STRING=HL1; VLAUE='LST' ← should be in data? VPTR: ORIGIN=NEXTI TERM: TERM(LOC)
PL1	LSG	LABEL: STRING LABEL=PL1 APTR: STRING=PE1;ORIGIN=AFTER VPTR: ORIGIN=NEXTI TERM: TERM(HE1)
PE1	ESG	LABEL: STRING LABEL=PE1 VPTR: ORIGIN=NEXTI TERM: VALUE=3;COUNTUNITS=PL1 instances
DEPT.DEP	RN	LABEL: STRING LABEL=DEPT.DEP APTR: STRING=DA1;ORIGIN=AFTER VPTR: ORIGIN=NEXTI TERM: VALUE=1,COUNTUNITS=Bytes
DEPT.MGR	RN	LABEL: STRING LABEL=DEPT.MGR APTR: STRING=DA1;DISPLACMENT SIZE=Ø VPTR: ORIGIN=NEXTI TERM: VALUE=2;COUNTUNITS=Bytes
DA1	ASG	LABEL: STRING LABEL=DA1 APTR: STRING=DL1;LAS NAME VALUE=EMP;DISPLACEMENT TERMINATOR=*; ORIGIN=START; APTR: STRING=DE2;DISPLACEMENT VALUE=2;ORIGIN=AFTER; DISPLACEMENT UNITS=Bytes VPTR: ORIGIN=NEXTI TERM: TERM(DEPT.MGR)
DL1	LSG	LABEL: STRING LABEL=DL1 APTR: STRING=DE1;DISPLACEMENT VALUE=6;ORIGIN=NEXTI; DISPLACEMENT UNITS=BYTES VPTR: ORIGIN=NEXTI TERM: TERM(EE2)

String	Type	Parameters
DE1	ESG	LABEL: STRING LABEL=DE1 VPTR: ORIGIN=NEXTI TERM: TERMINATOR='EF';FIELD USED=APTR
DE2	ESG	LABEL: STRING LABEL=DE2 VPTR: ORIGIN=START TERM: TERMINATOR='E';FIELD USED=VALUE

Figure 2-2 shows the Linear Address Spaces after they have been filled with the data values and structural information. The groupings above each line show which bytes belong to BEUs for which string instances. For example, the first 3 bytes of LAS EMP (Locations 00 through 02) contain all of the BEU for the first instance of the A-string EA1 that has not been factored into the catalog, namely an association pointer that gives the address of the next element on the first instance of E-string EE2 (Location 014). The next 4 bytes contain the three values for the role names associated with this instance of EA1.

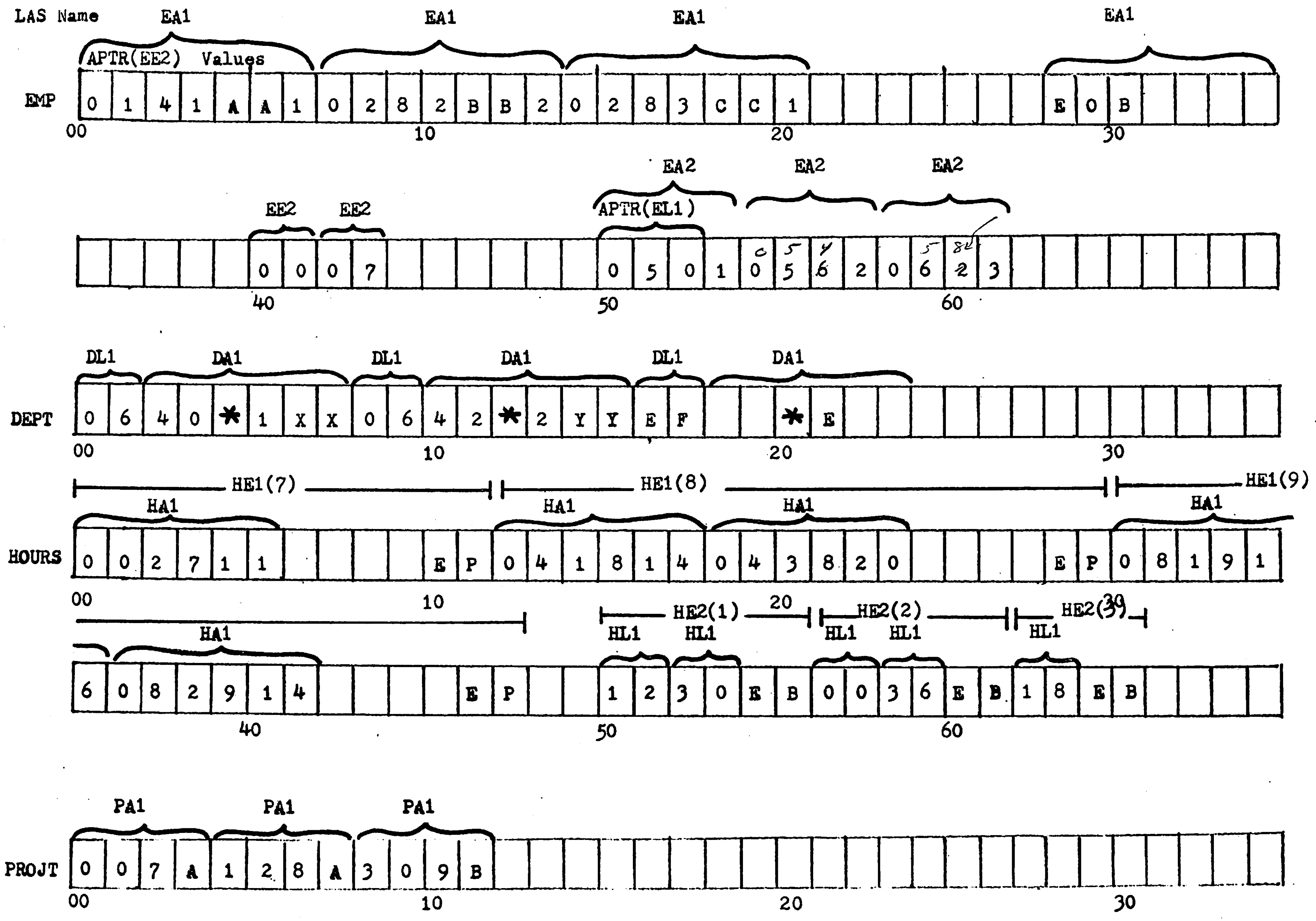

LAS Name
EA1
APTR(EE2) Values
EMP
0 1 4 1 A A 1 0 2 8 2 B B 2 0 2 8 3 C C 1
E 0 B
00
10
20
30
EE2
EA2
APTR(EL1)
0 0 0 7
0 5 0 1 0 5 6 2 0 6 2 3
40
50
60
DL1
DA1
DEPT
0 6 4 0 * 1 X X 0 6 4 2 * 2 Y Y E F
* E
HE1(7)
HE1(8)
HE1(9)
HA1
HOURS
0 0 2 7 1 1
E P 0 4 1 8 1 4 0 4 3 8 2 0
E P 0 8 1 9 1
6 0 8 2 9 1 4
E P
HE2(1)
HE2(2)
HE2(3)
HL1
1 2 3 0 E B 0 0 3 6 E B 1 8 E B
PA1
PROJT
0 0 7 A 1 2 8 A 3 0 9 B

FIGURE 2-2

3 - The Representation Independent Language

The Representation Independent Language (RIL) is the user's medium of interaction with the Entity Set Model. Its operations are set-theoretic in form, giving the user great flexibility in accessing and manipulating data without concern for the actual storage structures or accessing processes. The syntax of the RIL will be presented informally with brief explanations and examples. The examples will refer to the model developed in Section 2.

An Information Entity (I-entity) is a group of one or more data values, each of which is associated with a uniquely named attribute. An entity description from the ESM is termed an I-entity in the RIL.

An Information Set (I-set) is a uniquely named, structurally homogenous collection of one or more I-entities. I-sets are the entity description sets of the ESM. EMP, DEPT, HOURS, and PROJT are examples of I-sets. Since an I-set is structurally homogeneous, each of its I-entities will contain values associated with the same attributes. The names of these attributes are referred to as the template of the I-set, denoted by T(S), where S is the I-set name. For example the I-set EMP has T(EMP) = (EMP#,NAME,DEP) .

I-sets may be basic (part of the permanent data base), temporary (created and named by the user and defined only within the context of his query), or scratch (named by the user and defined within the context of a single statement).

The process of accessing data items in the ESM is expressed in the RIL as the selection of I-entities and I-sets according to the data values they contain. The selection criteria and the name of the I-set to which they are applied constitute the definition of a new I-set.

The basis for the selection process is the Entity Conditional Term (ECT), which is a boolean valued function of attribute values in the I-entities to which it is applied. Suppose we wish to select from the I-set EMP I-entities for employees in Department 1. The ECT (EMP.DEP = 1) could be used. It would have the value TRUE when applied to an I-entity in which DEP=1 and FALSE otherwise. This ECT is an example of a common type known as the Value Comparison (VALCOMP) ECT. An ECT is formed from 2 or more Entity Parameter Expressions (EPE) joined by comparison operators. The result of the comparison is the value of ECT.

An EPE is a function whose domain is one or more attribute values. In the ECT (EMP.NAME = AA) EMP.NAME is an EPE defined on the attribute EMP.NAME and whose output is merely the unaltered value of the attribute. AA is an EPE with a constant value. = is the comparison operator.

Other examples of EPEs combined into ECTs are:

EMP.EMP# $\leq$ 20

DEPT.MGR $\neq$ YY

\$5.50 x HOURS.HRS > \$100.00

LENGTH OF (EMP.NAME) $\geq$ 10

The complexity of EPEs is governed by the functions provided in the implementing system. Typically these could include standard arithmetic functions, (+,-,x,$\sqrt{}$, sin,...) and character string operations such as LENGTH, FIRST n CHARACTERS, and so on.

Several ECTs may be combined by boolean AND ($\wedge$) and OR ($\vee$) operators to form boolean valued Entity Conditional Expressions (ECE) so that if we wished to select employees in Department 1 with name AA we could write:

EMP.DEP = 1 $\wedge$ EMP.NAME = AA

A new I-set is defined by the Subsetting Operation which has the general form $S_t \longleftarrow S_s$: ECE where S_t is the target I-set and S_s is the source I-set from which entities are to be selected. For example, a temporary I-set (S1) containing entities for employees in Department 1 could be defined by the subsetting operation S1 $\longleftarrow$ EMP: S1.DEP = 1 . Notice that the ECE (S1.DEP=1) is a condition on an attribute value from the target set S1, not the source set EMP. The ECE is a condition which <u>all</u> elements of the target set must meet. This emphasizes that the subsetting operation is a definition of the target set and does not necessarily imply a procedure for its formation. The ECE in the example should be read as "The I-set S1 consists of I-entities which are also members of the I-set EMP and which satisfy the condition S1.DEP=1". The target set built up in this way is now available for use as a source set in future operations.

When accessing data, there may be situations in which selection of I-entities must be based on values associated with whole sets of, rather that individual, entities. Recall that the entity set HOURS contained entities that gave the hours worked by a given employee on a given project. In other words T(HOURS)=(EMP#,PROJ,HRS). Suppose, for example, we wanted the employee members of all employees who have worked a total of at least 30 hours on all projects. To retrieve these values we could form, sequentially, the I-sets:

S1 $\longleftarrow$ HOURS: S1.EMP# = 1

S2 $\longleftarrow$ HOURS: S2.EMP# = 2

•
•
•

Each temporary I-set contains all the "hours worked" I-entities for one employee. We could then sequentially access each I-entity in the temporary sets and add the HRS value to a cumulative total for the corresponding employee.

If this total exceeded 30 hours, that employee's number, name, etc. would be included in a target temporary set. For the convenience of the user the RIL contains constructions that will allow such complex and cumbersome procedures to be expressed in a single statement.

The expressions for stating selection criteria based on set values are analogous to those for handling entity values. Set Parameter Expressions (SPE) are functions which assign unique values to sets. For instance TOT(S1.HRS) could be the summing function required in the above example, returning a value that is a total of the specified attribute values for all I-entities in the named I-set. Other useful SPEs might be (where S1 is an I-set and A is an attribute in its template):

MAX(S1.A) returning the maximum value of the attribute over all entities in the set;

AVG(S1.A) returning the average value; and

COUNT(S1) returning the number of I-entities in the set. Other SPEs can be defined as needed.

VALCOMP Set Conditional Terms (SCT) are formed by using comparison operators to connect SPEs. $\text{TOT(S1.HRS)} \geq 30$ is a SCT which would have the value TRUE if the total hours worked by Employee 1 were at least 30.

Set Conditional Expressions (SCE) are boolean combinations of SCTs. The SCE $(\text{TOT(S1.HRS)} < 20) \wedge (\text{COUNT(S1)} > 2)$ would have the value TRUE if Employee 1 had worked less than 20 hours on more than 2 projects.

The SCE provides a means of selecting sets according to a given criterion, but there must be a means of dynamically producing the sets to which it is to be applied. In the RIL this accomplished by defining a new type of entity conditional term, the FOR ECT, which has the general form $\text{FOR } [S_c :: \text{SCE}]$

where S_c is a scratch I-set. The ECT has the value TRUE for a given instance of S_c when the SCE is TRUE for that instance. The FOR ECT can then be used in a subsetting operation with the definition of S_c dependent on the I-entities of the source I-set. To form the subset of employees working more than 30 hours we could now write

$$\text{S2} \longleftarrow \text{EMP: FOR } \left[(\text{S1} \longleftarrow \text{HOURS: S1.EMP\#=S2.EMP\#})::(\text{TOT(S1.HRS)} \geq 30)\right]$$

This expression defines the I-set S2 as consisting of I-entities from EMP. Each element of S2 satisfies the condition that if a scratch set S1 is formed from HOURS so that each element of S1 has EMP# equal to the EMP# in the element of S2 then the total of the HRS values for elements of S1 will be ≥ 30.

Several additional types of ECTs and SCTs are provided for special purposes. The EXIST ECT has the form $\in(S_i \longleftarrow S_j : \text{ECE})$ and has the value TRUE if the I-set S_i is non-null, that is if there exists an entity in S_j that meets the condition given by the ECE. The EXIST ECT is equivalent to the expression $\text{FOR } [S_i \longleftarrow S_j\text{:ECE::COUNT}(S_i) \geq 1]$.

The EXCLUDE ECT is the logical complement of the EXIST ECT. It has the form $\notin(S_i \longleftarrow S_j : \text{ECE})$ and is equivalent to $\text{FOR } [S_i \longleftarrow S_j\text{:ECE::COUNT}(S_i)=0]$

The ALL SCT has the form $\text{ALL}(S : \text{ECE}_i :: \text{ECE}_j)$. It has the value TRUE if and only if ECE_j is TRUE for all entities in S for which ECE_i is TRUE. For example ALL(EMP:EMP.DEP=1::EMP.EMP#<2) would be FALSE since not all of the employees in Department 1 have EMP# less than 2. (See page38) If ECE_i is not present, the SCT becomes a test of whether all entities in S meet the criteria expressed in ECE_j.

In a model such as the ESM where related data values are stored in different data sets it may often be necessary to perform set intersections to access all the desired data pertaining to a given real world object. The intersection of two I-sets could be accomplished using the FOR ECT and subsetting operation as in the above example. However, because of the frequency of this operation a special expression for it has been provided.

The I-aggregate is a set of I-entity combinations. It is defined by an I-Structure Specification Expression (SSE) having the form:

$$[S_i \mid S_j : \text{ECE}]$$

Each element of the I-aggregate contains a pair of I-entities, one from S_i and one from S_j, that meets the criteria stated in the ECE. For example, to link information on employees with records of their hours worked, it is necessary to intersect the I-sets EMP and HOURS on the basis of EMP#. This would be expressed by $[\text{S1} \longleftarrow \text{EMP} \mid \text{S2} \longleftarrow \text{HOURS} : \text{S2.EMP\#=S1.EMP\#}]$

Since in this example there are several I-entities in HOURS for each employee, each I-entity from EMP will appear in several elements of the I-aggregate. If this is undesirable for the application at hand, the operator α can be employed as follows:

$$[\text{S1} \longleftarrow \text{EMP} \mid \alpha(\text{S2} \longleftarrow \text{HOURS:S2.EMP\#} = \text{S1.EMP\#})]$$

which indicates that each element of S1 is to be linked with a set of all elements in S2 for which the EMP#s match. In general the α operator causes the intersection of two I-sets to result in a hierachical structure with single elements from one set linked to groups of elements from the other.

To illustrate I-aggregate and the α operator, consider the two I-sets

EMP	HOURS
EMP#/1	EMP#/1,PROJ/8
EMP#/2	EMP#/1,PROJ/9
EMP#/3	EMP#/2,PROJ/7
	EMP#/2,PROJ/9
	EMP#/3,PROJ/8

The I-aggregate resulting from the SSE $[S1 \leftarrow EMP \mid S2 \leftarrow HOURS : S2.EMP\#=S1.EMP\#]$ would be

```
EMP#/1  -  EMP#/1,PROJ/8
EMP#/1  -  EMP#/1,PROJ/9
EMP#/2  -  EMP#/2,PROJ/7
EMP#/2  -  EMP#/2,PROJ/9
EMP#/3  -  EMP#/3,PROJ/8
```

The SSE $[S1 \leftarrow EMP \mid \alpha(S2 \leftarrow HOURS:S2.EMP\#=S1.EMP\#)]$, utilizing the operator would result in

$$EMP\#/1 - \begin{Bmatrix} EMP\#/1,PROJ/8 \\ EMP\#/1,PROJ/9 \end{Bmatrix}$$

$$EMP\#/2 - \begin{Bmatrix} EMP\#/2,PROJ/7 \\ EMP\#/2,PROJ/9 \end{Bmatrix}$$

$$EMP\#/3 - \{EMP\#/3,PROJ/8\}$$

While the I-aggregate provides the links to effect set intersection, it is not itself an I-set and so cannot be used in further processing. An I-set can be created from an I-aggregate, however, using the Derivation Operation. This operation has the form:

$$S_t(a_1,a_2,\ldots,a_n) \Leftarrow [SSE] \quad VAT_1,VAT_2,\ldots,VAT_n$$

where $a_1,a_2,\ldots,a_n$ define the template of the new set S_t, and SSE defines some I-structure, possibly and I-aggregate. $VAT_1,VAT_2,\ldots,VAT_n$ are Value Assignment Terms written in the form $a_i := VEXP_i$. $VEXP_i$ is a Value Expression, a function of attribute values in the I-structure from which the new set is being derived. Each attribute in the $T(S_t)$ will have a corresponding VAT.

The following examples illustrate the derivation operation:

The set of names of all employees is defined by:

$$S2(NAME) \Leftarrow [S1 \leftarrow EMP] \quad NAME:=S1.NAME$$

A list of all employees and their assigned managers is given by:

$$S3(NAME,MGR) \Leftarrow [S1 \longleftarrow EMP \mid S2 \longleftarrow DEPT: S1.DEP=S2.DEP]\ NAME:=S1.NAME, MGR:=S2.MGR$$

Using the derivation operation, the set of employees with over 30 hours worked can be expressed in an alternative way to that on page 51:

$$S4 \longleftarrow (S3(EMP\#) \Leftarrow [S1 \longleftarrow EMP \mid \alpha(S2 \longleftarrow HOURS: S2.EMP\#=S1.EMP\#)]\ EMP\#:=S1.EMP\#): TOT(S2.HRS) \geq 30$$

If no information about the employees besides their EMP# is desired, this form permits the elimination of uneeded data values from the target set.

Most of the RIL elements discussed so far are oriented toward retrieval of stored data. But the language also provides for insertion, update and deletion as well as input/output operations.

Insertion is essentially the inverse of the subsetting operation. It is prescribed by an expression of the form

$$S_s \longrightarrow S_t : ECE$$

The effect of insertion is to copy into the target set S_t those I-entities from the source set S_s for which the ECE is TRUE. If the ECE is omitted, all elements of S_s are added to S_t. Insertion requires that the templates of the target and source set be identical.

To illustrate insertion, suppose S1 contains I-entities for new employees and has the same template as EMP. In order to add to the EMP set those employees that are assigned to departments 1 or 2 we can write

$$S1 \longrightarrow EMP: S1.DEP \leq 2$$

Note that the ordering of the I-entities in EMP is not a matter of concern for the user in this example. In fact ordering is transparent to the user in all phases of the RIL, a point indicative of the data independence that it provides.

Update is indicated by an expression of the form:

$$S_s \Rightarrow [S_t : \mathrm{ECME}]\ \mathrm{VAT}_1, \mathrm{VAT}_2, \ldots$$

where S_t is the set to be updated, S_s contains updating information and $T(S_s)$ and $T(S_t)$ have at least one attribute in common. ECME is an I-entity Conditional Matching Expression which has the same form as an ECE and is used to select elements of S_t to be updated based on a match of the common attribute values in S_t and S_s. $\mathrm{VAT}_1, \ldots$ are of the form $a_i{:=}\mathrm{VEXP}_i$, and they describe how attribute a_i in the I-entity selected from S_t is to be modified. For example, if we wish to change the department managers, and I-set S1 could be formed with T(S1)=T(DEPT), the I-entities of S1 containing the new manager names. The update expression would then be:

$$\mathrm{S1} \Rightarrow [\mathrm{DEPT{:}DEPT.DEP{=}S1.DEP}]\ \mathrm{MGR{:=}S1.MGR}$$

If an update operation requires no input, the expression would be written:

$$(\ ) \Rightarrow [S_t : \mathrm{ECE}]\ \mathrm{VAT}_1, \mathrm{VAT}_2, \ldots$$

For instance to move employees in Department 2 to Department 3 we use:

$$(\ ) \Rightarrow [\mathrm{EMP{:}EMP.DEP{=}2}]\ \mathrm{DEP{:=}3}$$

Deletion is expressed by:

$$\backslash\backslash \Rightarrow [S_t : \mathrm{ECE}]$$

where ECE is used to locate the I-entities in S_t that are to be removed and \\ is a special deletion symbol. To remove the record of hours worked by employee 2 on project 9 the expression is:

$$\backslash\backslash \Rightarrow [\mathrm{HOURS{:}HOURS.EMP\#} = 2]\ \mathrm{HOURS.PROJ} = 9$$

If the user wishes to have a temporary I-set appear as output from the model, this can be done by enclosing the set name in \ \ at the time it is defined, giving an expression of the form: $\backslash\, S_t \backslash \leftarrow$(Set definition expression).

For example $\backslash S1 \backslash \longleftarrow EMP:S1.DEP=1$ would cause the records for employees in Department 1 to be output.

For input operations it is assumed that data to be input is assembled in a file external to the ESM. Each record of this file contains data values that are to be placed in a single I-entity. The Input Statement has the form:

$$S_t \longleftarrow \backslash(a_1,a_2,\ldots,a_n)\backslash \ : ECE$$

S_t is a temporary I-set that is to receive the new values. $(a_1,a_2,\ldots,a_n)$ are attribute names that specify the template of S_t. The ith field in a record of the external file will hold the value assigned to a_i in an I-entity of S_t. The ECE permits the conditional selection of records from the external file.

To input a new set of employee records we would write:

$$S1 \longleftarrow \backslash T(EMP)\backslash$$

If the external file held "hours worked" records and we wished to input only those pertaining to project 7, the expresseion would be:

$$S2 \longleftarrow \backslash T(HOURS)\backslash : S2.PROJ=7$$

4 - A Representation Dependent Language

The Representation Dependent Language (RDL) provides a means for stating, in terms of the string model, procedures which, when executed, will result in the desired operations on the data base. It is necessary that the RDL have the capability of expressing procedures that will implement any statement in the Representation Independent Language (RIL). Thus it must provide for retrievals, insertions, deletions, and changes of data items. In addition the RDL will permit the specification of procedures which alter the structure of the string model. One possible RDL is proposed here.

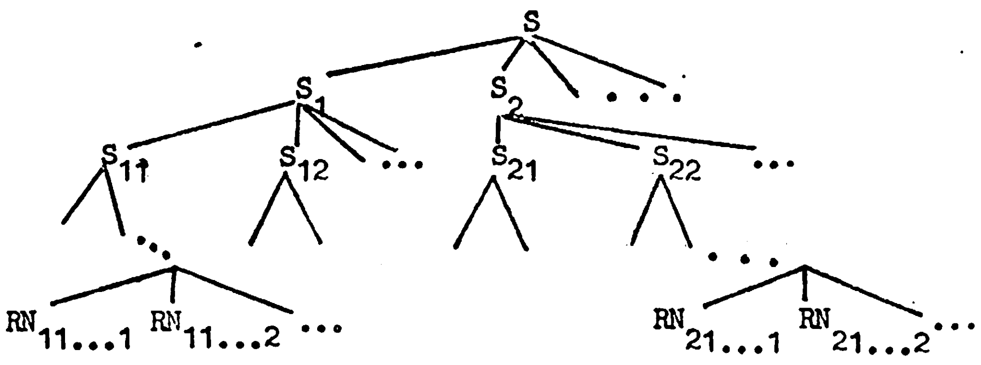
Elements of the RDL

String Structure Expressions (SNGSE)

An SNGSE will have the general form

$$ES.S.(S_1,S_2,\ldots).(S_{11},S_{12},\ldots);(S_{21},S_{22},\ldots).\;\ldots\;.(RN_{11\ldots1},RN_{11\ldots2},\ldots);\ldots$$
$$(RN_{12\ldots1},RN_{12\ldots2},\ldots);\ldots$$

where ES is an entity set name, S, S_1, S_{11}, etc. are string names and $RN_{11\ldots1}$, etc. are role names. Different levels of the string structure are seperated by single periods. Groups of elements at the same level are seperated by semi-colons. Elements enclosed in parentheses comprise the Exit List of a string at the next higher level, so that S has EXL=$(S_1,S_2,\ldots)$, S_1 has EXL=$(S_{11},S_{12},\ldots)$ and S_2 has EXL=$(S_{21},S_{22},\ldots)$, etc. If the above structure were viewed as a tree it would have the form



Thus the number of subscripts an element has indicates the level at which it will be found and the subscripts themselves identify the parent strings. For instance S_{ijk} would appear at the 4th level of the structure and would have EXL = $(S_{ijk1}, S_{ijk2}, S_{ijk3}, \ldots)$. Referring to the example in Section 2, the string structure containing the strings DE1,DL1 DA1,EE2, and EA1 would be represented by the SNGSE

DEPT.DE1.(DL1).(DA1,EE2).(DEP,MGR);(EA1).(EMP#,NAME,DEP)

String Declaration Statement - CREATE SNGSE_D

where SNGSE_D is a string structure expression which has been modified as follows: 1) Each component may be followed by an expression in [ ] that specifies values for parameters appropriate to the general string type of the component. 2) If the component is a role name it may be enclosed in ⟨ ⟩ to indicate that it is an identifier for the entity set in which it appears. The execution of a CREATE statement causes the generation of catalog entries for the strings in the expression. To illustrate, consider the statement

CREATE TE1.TL1 [MC=(TA1.EMP#=EA1.EMP#)] .(TA1,EA1).(⟨PROJ⟩,EMP#)

which establishes a structure consisting of a new E-string TE1 which will connect instances of a new L-string type TL1. The exit list of TL1 includes a new A-string TA1 and an already existing A-string EA1. Instances of these strings are linked on the basis of matching values for the role name EMP#. TA1 has EXL=(PROJ,EMP#) with PROJ as the identifier for the new entity set. Role names need not be given for EA1 since it is already in the catalog. In general, the effect of this statement is to create new string types. Instances of these types will be formed at a later stage by assignment of values to the string and role names listed here.

String Acquisition - GET S

where S is a string,(possibly subscripted), that is on the Entry List. (It is assumed that each string for which ON=ENTRY is listed, together with the address of its first BEU, in an accessible table called the Entry List). The GET statement initiates at the encoding level the pointers and other mechanisms necessary to follow the linked lists which constitute paths through the data beginning at the entry location specified here. At the string level the effect of this statement is to retrieve the instance of the string S. (There will be only one instance of S since it is on the Entry List.) Thus S becomes the Present String and the Current String of its type as described below.

Present String - The name and LAS address of the last string instance to be retrieved is placed on a push-down stack. The top entry on this stack is the Present String (PS).

Current Strings - A stack will be maintained for each type of string retrieved, and the name and LAS address of each instance retrieved will be placed on its appropriate stack. The top entry on a stack will be the Current String for that type, designated by CS(String Name). e.g. CS(EA1).

String Search - DO FOR S: C [B]

where S is the name of a string on the exit list of the PS (Present String), C is a conditional expression that selects instances of S, and B is a block of RDL statements. B will be executed for each instance of S that is selected. The expression C may be a boolean function defined on data values that can be accessed by at most 3 pointer retrievals. That is if S is an A-string the values of role names on its exit list may be tested (2 pointer retrievals); or, if S is an L-string which has an A-string as the first element of its exit list, the values of role names on that A-string may be tested (3 pointer retrievals).

(While deeper searches for data values to be tested could theoretically be allowed, they are neglected at this point because of the complexities they would cause and because any operation for which they would be needed could be performed by using temporary strings and multiple search passes.) C may also be a boolean condition on some function of a count of the instances of S that have been retrieved, destignated COUNT(S). The word ALL will indicate that every instance of S on the PS is to be selected. When an instance of S is selected it is placed on the PS stack (and thus becomes the PS) and on the CS stack for its string type. The [ ]define the scope of the search statement within which only data values accessible from one of the CSs by no more than 3 pointer retrievals are available. When execution of B is completed, the PS is used to find the next instance of S to which C is applied. When the pertinent set of S instances is exhausted, PS is restored to the value it had before the search statement was encountered. Search statements may be nested to any level.

Assignment Statement - ASSIGN SNGSE_A

where SNGSE_A is a string structure expression that has been modified as follows: Each component may be followed by an assignment symbol $\longleftarrow$ and an expression. If the component is a role name the expression will be a function of data values that can be retrieved from current strings. If the component is a string name the expression must be an already existing string or string structure of the same type (i.e. A-string, E-string or L-string) in which case the assignment constitutes a link to that structure. Execution of an assignment statement creates an instance of the structure named in the SNGSE. If the components of the SNGSE are string names which appear in search statements within whose scope the assignment statement appears, then the

SNGSE is taken to be a specific instance of the named structure composed of the current strings. In this case the assignment represents a modification of the designated instance. The modification could be deletion, symbolized by ϕ . The assignment of data values to OUT denotes output from the system.

To illustrate RDL procedures consider the following example. Suppose the structure to be searched has the form ES1.E1.(A1).(RN1,RN2) . We wish to select all instances of A-string A1 for which role name RN2 has the value x, and we will place the values for RN1 for the selected instances in a new structure specified by the statement

```
CREATE ES2.E2.(A2).(RN3)
```

The remainder of the RDL procedure will be

```
GET E1
DO FOR A1: RN2=x
   [ASSIGN A2.RN3 <—— RN1]
```

The assignment statement will be executed once for each instance of A1 that contains the desired value of RN2, and each time it is executed a new instance of A2 is created with RN3 being assigned the current value of RN1.

Now, taking the original string structure in the above example and the new structure created by the procedure, suppose we wish to form a link between instances of A1 and A2 based on a matching value for RN1 and RN3. The RDL procedure to accomplish this is

```
CREATE ES1.E3.L1.(D1,D2)
GET E2
DO FOR A2: ALL
   [GET E1
    DO FOR A1: A1.RN1=A2.RN3
       [ASSIGN L1.(D1<—— A2,D2<—— A1)]]
```

(D1 and D2 are dummy string names. Actual names will be supplied in the assignment statement.)

5 - Translation of RIL Statements into RDL Procedures

Given an RIL statement that describes a certain operation to be performed, we wish to form a group of RDL statements which describes the same operation in terms of the String Model. In addition, since there may be more than one group of statements that could represent the operation, we wish to choose the RDL representation that is in some sense the best.

5.1 - Choosing an Access Path

An RIL statement will specify certain data values that are to be operated upon and it is necessary to choose a path through the String Model from an entry point to the role names whose associated data values will be those that are required. The process of choosing a path has two phases: Path Enumeration - listing all possible paths from entry points to each role name; Path Selection - choosing from all the paths listed the ones which can be most efficiently used in performing the desired operation.

Path enumeration can be performed by taking each role name from the parse of the RIL statement, finding its ON list from the catalog and tracing back to entry points using all elements of each ON list encountered. This produces a set of paths for each role name. For example the RIL statement $T \longleftarrow EMP : T.DEP = \uparrow \wedge T.EMP\# < 3$ requires access to data values for role names EMP/DEP and EMP/EMP#. The ON lists retrieved by starting with the catalog entry for EMP/EMP# are:

EMP/EMP#	ON=EA1,EA2
EA1	ON=EE1,EE2
EA2	ON=EL1
EE1	ON=ENTRY
EE2	ON=DL1
EL1	ON=EE3
EE3	ON=ENTRY
DL1	ON=DE1
DE1	ON=ENTRY

Thus the possible paths to EMP/EMP# are:

(1) EMP/EMP# ← EA1 ← EE1 ← ENTRY
(2) EMP/EMP# ← EA1 ← EE2 ← DA1 ← DL1 ← DE1 ← ENTRY
(3) EMP/EMP# ← EA2 ← EL1 ← EE3 ← ENTRY

Note that when an L-string occurs in a path all elements of its exit list are included also to show the different string types that will be accessed in a search of the L-string. The paths to instances of EMP/DEP are:

(4) EMP/DEP ← EA1 ← EE1 ← ENTRY

(5) EMP/DEP ← EA1 ← EE2 ← DA1 ← DL1 ← DE1 ← ENTRY

This completes the process of path enumeration.

Path selection requires evaluation of each path that has been generated with reference to the operations specified in the RIL statement. This process should reveal which paths are essential to the operation and indicate, if alternatives exist, which is the most efficient. Several heuristics are suggested here to guide this selection.

5.11 - Role Name Inclusion

It is possible that a single path may provide access to several of the required role names. If so it may be preferred because the needed data can then be accessed by a single search procedure. If there is no single path to two role names whose values must be accessed simultaneously, then this operation must be performed, at least conceptually, by forming temporary entity sets to hold the values from the search of each path, and then taking the intersection of the temporary sets. In the extreme case this may reduce to an exhaustive search of several entity sets. Thus it seems reasonable to favor a path that accesses as much of the needed data as possible in a single search. This criterion is applied to a path by checking the exit list of each A-string for the presence of the desired role names. While in a strict sense only the role names on the terminal A-string are on the path in that their BEUs contain pointers in the chain from entry point to final data value, the role names on non-terminal A-strings can be accessed with relative ease once the A-string is reached. Thus a role name will be considered to be available on any path in which its A-string appears.

To illustrate this heuristic we can apply it to the five paths generated above. By the convention that the presence of an A-string implies the availability of all role names on it, paths (1) and (4) are equivalent as are (2) and (5), and so (1) and (2) will be dropped from consideration. Checking the exit lists of each A-string we find the role names available on each path are:

(3) EMP/EMP#

(4) EMP/EMP#, EMP/NAME, EMP/DEP

(5) EMP EMP#, EMP/NAME, EMP/DEP, DEPT/DEP, DEPT/MGR

Thus paths (4) and (5) contain all the data values that need to be accessed and will be preferrable to (3) which contains only one role name.

5.12 - A-string Grouping Conditions

The presence of conditional terms (ECTs, SCTs) in an RIL statement indicates that data values are to be selected from a given set of values according to some criteria defined on the values themselves or on the values of related attributes. At worst this selection can be carried out by a linear search of the entity set, testing values as they are encountered. There may, however, be paths available containing strings that effectively partition an entity set according to the conditional criteria specified. If this is the case, then search time can be saved by limiting the search to a subset of the target values. Such partitioning is typically done according to the

Set Selection Criteria of E-strings. This parameter can be checked each time an E-string is found on a path. If the role name in the SSC is the same as one of those appearing in an ECT or SCT, the path containing the E-string may be preferred.

An RIL statement may also require value selection based on comparison of data values from different entity sets. This could occur in a conditional expression or an I-aggregate construction. The obvious string structure to support this type of access is an L-string linking elements of the subject entity sets. Thus it is beneficial to check the Match Criteria of L-strings on a path for the presence of role names from different entity sets that are involved in RIL comparisons.

In the above example the choice of entities from the I-set EMP is conditional on values of EMP/EMP# and EMP/DEP. Checking the SSC for EE2 we see that this E-string partitions the set of EA1 string instances according to values of EMP/DEP. Thus path (5) which contains EE2 is favored over (4) by this criterion.

5.13 - String Instance Count

When accessing a particular subset of data it will probably be necessary at some point to search through a set of string instances and choose one meeting certain conditions that are known to lead to the target data. For example in using path (5) to reach entities for employees in Department 1 it is necessary to search the instances of the L-string DL1 on E-string DE1 to find the one that leads to values associated with Department 1. Using path (4) it would be necessary to examine each instance of EA1 on EE1, testing the value of EMP/DEP. When alternative search paths are available it may be relevant to check what instance searches are invloved in each and the relative sizes of the sets to be searched. This information may be available as termination information for E-strings at the encoding level or could be estimated from periodically updated instance counts of the important string structures. In the above example, if the nature of the data were such that there were more departments than employees, it might be more efficient to use path (4) and search the set of EA1 strings than to search the larger set of departments. If, as is more likely, the cardinality of EMP is greater than that of DEPT, then this rule gives added incentive for choosing path (5).

5.14 - Path Length

If several paths are viewed as equal according to the above three criteria, it may be of some advantage to choose the path containing the fewest string types, since following this path will involve the fewest pointer retrievals and address calculations.

5.2 - Forming the RDL Procedure

If complete flexibility is to be achieved, the DIAM must be able to service any legal RIL request. It is likely that the most frequent requests will have been anticipated in the design of the string model and that the strings which lead to the data values they specify are already available. In these cases the RDL procedures for processing the requests will involve no more than following the selected access path from the entry point to the desired data values. If any branching decisions must be made, the data values on which to base the decisions will always be immediately available. For instance, suppose the RIL statement Q1 ← EMP: Q1.DEP=1 has resulted in selection of the path EA1 ← EE2 ← DA1 ← DL1 ← DE1 ← ENTRY . One reason this path was chosen is that the match condition on the L-string DL1 is based on the value of DEP. (See Section 2) The instances of DA1 representing departments are explicitly linked to groups of EA1 instances representing employees in the same department. This structure was, of course, intended to support just such a query. The RDL procedure to implement it is a sequence of search statements, one for each string in the path. The FOR condition in the statement where a single instance of DL1 must be selected will be precisely the same as the condition in the Entity Conditional Term of the RIL statement. The procedure will be:

```
GET DE1
DO FOR DL1: DA1.DEP=1
   [DO FOR DA1: ALL
       [DO FOR EE2: ALL
           [DO FOR EA1: ALL
               [ASSIGN OUT ←(EMP#,NAME,DEP)]]]]
```

In such cases the formation of the RDL procedure appears to be relatively straightforward.

In those instances in which requests are made that the string structure was not designed to process, more complex RDL procedures may well be required. In addition, any algorithm for generating the RDL expressions may have to choose among alternative procedures, and so must have access to some criteria of efficiency. In the following, some of the aspects of this problem are explored.

All data base transactions requested by users involve the accessing of particular data items; that is, the generation of physical device addresses at which data values for particular role names associated with given entity descriptions are located or are to be located. A-strings are defined in such a way that for a given A-string type there will usually be a one-to-one correspondence between A-string instances and entity descriptions, and the structure of the A-string (its defining set) is relatively stable. Thus at the string level it is convenient to restate the basic transaction problem as one of accessing subsets of A-string instances. Once an A-string has been retrieved, it is a straightforward operation to locate the role names and data values associated with it. Thus a request to access data for Employee 1 will be interpreted as a request to retrieve the A-string instance which has in its defining collection the role name EMP# with associated value 1.

In a typical String Model several A-string types, or sets of A-string instances, will be defined. In general, there will be at least one set of A-string instances for each entity description set. Some description sets may be represented by more than one A-string type, each type being defined by different combinations of the role names from the description set. The satisfaction of a user query may require the retrieval of A-string instances from one or more of these sets. For example S1 ⟵ EMP: S1DEP = 1 involves accessing instances of one A-string type, whereas

S2 ⟵ EMP: FOR [S3 ⟵ DEPT: S3.DEP::ALL(S3:S3.MGR=XX)]

requires that instances of two different A-strings be retrieved, one linking the role names in the description set EMP and the other linking the role names in DEPT. In addition, there may be occasions in which the set of instances of a given A-string type is partitioned into subsets by instances of a higher-level string. The E-string EE2, for example, partitions the set of A-string EA1 instances according to department, forming one subset of instances of EA1 representing Employees 1 and 3 and another from the instance of EA1 representing Employee 2.

As stated, the process of accessing means the generation of addresses. In the DIAM this means the retrieval of pointers, either explicit addresses from the data stream or implied pointers from the catalog. Because of the list structure used in the model, the pointer to any given element (string instance, role name, or data value) can be obtained from the Entry List or by retrieving the preceding element in some list (string) which contains the target element. The predecessor will supply the required pointer. In practice, A-strings will never appear on the Entry List..To place the address of many A-string instances on the Entry List would waste the advantage of defining higher level strings over the set of A-strings. Thus we will assume that pointers to A-strings will only be found by tracing some higher level string and decoding the pointers as they are encountered.

From the two preceding paragraphs we see that all string models possess a number of basic sets of A-string instances. These sets are distinguished by the fact that they are all included within the scope of a single instance of a higher level string, or in other words, that each A-string instance in the set contains a pointer (APTR) to the next instance in the set. The first instance is pointed to by the higher string instance (VPTR). We will call such a set of A-string instances a linked set. The concept of a linked set will be useful in choosing an efficient implementation for a given transaction request.

If the RIL query to be implemented involves A-string instances from only a single linked set, then the natural RDL procedure to satisfy it is a single search statement in which the FOR clause gives the condition for selection of instances from the linked set. For example, consider a request for the Department managed by Manager XX, given by the RIL statement S1 ⟵ DEPT: S1.MGR = XX. As shown in Chapter 2, the A-string DA1 is defined over the entity descriptions for departments, and a single instance of a higher level string (DE2) is defined over DA1. Thus all instances of DA1 form a linked set, and this is the only linked set involved in the query. These string names would have been found by the path selection methods of Chapter 5, which would produce the path DA1 ⟵ DE2 ⟵ ENTRY. We can then form the RDL procedure

```
GET DE2
DO FOR DA1: MGR = XX
  [S1.SA1.DEP ⟵ DEP]
```

If the requested transaction involves A-string instances from more than one linked set, then some type of set intersection must be performed. Set intersection implies, at least conceptually, the simultaneous accessing of elements from different sets, in this case A-string instances from two or more linked sets. Since the RDL presented here does not provide for parallel operations, the problem of implementing a set intersection becomes one of choosing the sequential process that most efficiently accomplishes the desired retrievals from the several linked sets in question. There are three procedures that could be used for set intersection.

Alternating Search Method- Using this method, an A-string instance is chosen from the first linked set (assume two sets are to be intersected). The search of the first set is then suspended with the identity of the current string being saved, and a second search routine for the second set is entered. The element of the second set that matches the current element of the first set is selected and the intersection is performed for those elements. The second search routine is then exited and the first routine continued to the next element.

Suppose that two paths have been selected

$$A1 \leftarrow E1 \leftarrow \text{ENTRY} \qquad (1)$$

$$A2 \leftarrow E2 \leftarrow A3 \leftarrow L1 \leftarrow E3 \leftarrow \text{ENTRY} \qquad (2)$$

and that the linked sets to be intersected are the set of all instances of A1 and the subset of instances of A2 which defines a given instance of E2. The sequence of RDL statements needed to perform this intersection by the Alternating Search method would be:

```
GET E1
DO FOR A1: ALL
   [GET E3
    DO FOR L1: A3.R1=x                    (Selects desired instance of E2)
       [DO FOR A3: ALL
           [DO FOR E2: ALL
               [DO FOR A2: A2.R1=A1.R1      (Intersection match)
                   [ASSIGN OUT ←A1,A2]]]]]
```

<u>Temporary Set Method</u> - This method consists of accessing one of the linked sets and copying the desired A-string instances from it into a set of temporary A-strings with a simpler structure. The second linked set is accessed and all instances in it are placed in a second temporary set. The temporary sets are then intersected using the Alternating Search method.

The RDL statements needed to implement this method for the example given above are:

```
CREATE TS1.TSE1.TSA1.(EXL same as for A1)
CREATE TS2.TSE2.TSA2.( "    "   "   "  A2)
GET E1
DO FOR A1: ALL
   [ASSIGN TSA1 ←A1]
GET E3
DO FOR L1: A3.R1=x
   [DO FOR A3: ALL
       [DO FOR E2: ALL
           [DO FOR A2: ALL
               [ASSIGN TSA2 ←A2]]]]
```

The above aprocedure forms the temporary sets. To carry out the intersection we need:

```
GET TSE1
DO FOR TSA1: ALL
  [GET TSE2
   DO FOR TSA2: TSA1.R1=TSA2.R1
     [ASSIGN OUT ←— TSA1,TSA2]]
```

<u>New String Method</u> - Since the main function of an L-string is to link elements from different sets, they are ideally suited for use in set intersections. If an L-string is defined that links instances of one A-string type to those of another, then this intersection has in a sense been wired into the String Model. A pair of A-string instances can be retrieved simply by following pointers from one to the other, and in fact we will say that each L-string defines a separate linked set. If the set intersection has not been built into the permanent structure by L-strings, then the RDL permits them to be added dynamically as new strings. The effect of this is to superimpose over the two old linked sets, an array of new linked sets, each containing a pair of A-string instances, joined according to the condition for the intersection. The method now reverts to the case of a single linked set, since as each set is retrieved all the required information can be obtained from it without reference to any other linked set.

It should be noted that when this method is used pointers to implement the newly formed links may have to be inserted into the previously existing data stream. Specifically, when two linked sets of A-strings are being joined by a new L-string, BEUs for A-string instances in the first set will have to have an extra APTR for the L-string. This may cause significant complexities at the encoding level.

The sample intersection described above, if performed by the New String method, would require the following RDL statements.

First to form the new string structure

```
CREATE E3.L2.(D1,D2)
GET E1
DO FOR A1: ALL
  [GET E3
   DO FOR L1:A3.R1=x
     [DO FOR A3: ALL
       [DO FOR E2: ALL
         [DO FOR A2: A2.R1=A1.R1
           [ASSIGN L2.(D1 ←A1, D2 ← A2)] ] ] ]
```

Then to search the new structure

```
GET E3
DO FOR L2: ALL
  [ASSIGN OUT ←A1
   DO FOR A1: ALL
     [ASSIGN OUT ← A2] ]
```

The problem now becomes one of choosing the best of the above methods to handle the given transactions. At this point, a reasonable criterion for "best" would seem to be the number of pointer retrievals that must be made. Costs of storage cannot be dealt with without further consideration of the physical implementation to be used, and catalog complexity is not well enough defined to serve as a criterion. Thus we will assume that the best accessing method is the one that requires the minimum number of pointer retrievals.

Before exploring the costs of the three methods described, we define the following functions.

$i(S)$, where S is a string type, gives the number of instances of S currently in the data base. For example if there are 100 employees, each represented by an instance of an A-string EA1, then $i(EA1) = 100$.

c(S) , where S is an A-string, gives the number of instances of S that must be retrieved to satisfy the current RIL request. For example if the current query calls for information on one employee, c(EA1) = 1 for that request. If information is desired on all employees or on all employees meeting a certain criterion, then $c(EA1) = i(EA1)$.

$\ell(N)$, where N specifies an access path from an entry point to an A-string, gives the number of string instances that must be retrieved when the path is followed, exclusive of the first A-string instance. For the path $A1 \leftarrow E1 \leftarrow L1 \leftarrow E2 \leftarrow \text{ENTRY}$ (1)

$\ell(1) = 3$

Consider the case in which two sets are to be intersected that cannot be reached by the same access path. In other words during the path selection process it was found that no single path contained all the required role names. Say the two paths selected are

$$A1 \leftarrow S1 \leftarrow S2 \leftarrow S3 \leftarrow \text{ENTRY} \qquad (1)$$

$$A2 \leftarrow S4 \leftarrow S5 \leftarrow \text{ENTRY} \qquad (2)$$

The linked sets to be intersected are the sets of all instances of A-strings A1 and A2, and S1, S2, etc. are higher level strings. Note that S1 and S4 must be E-strings, since we have specified that there are only two linked sets. The intersection of A1 and A2 will result in a set of pairs of A-string instances which have matching values for one of their role names. Assume that the query being processed requires that all instances of A1 are to be matched, so that $c(A1) = i(A1)$.

The Alternating Search method would require that path 1 be followed until S1 is retrieved, a process involving 3 pointer retrievals. Retrieving each instance of A1 will require 1 pointer retrieval, so the number of retrievals needed in searching path 1 will be $3 + \dot{\imath}(A1)$. Similarly the pointer retrievals needed to search path 2 are $2 + \dot{\imath}(A2)$. Since path 2 must be searched once for each instance of A1 until a matbh is found, the average number of retrievals for a search of path 2 will be $2 + \frac{1}{2}\dot{\imath}(A2)$. The number of pointer retrievals needed for the entire intersection process is then

$$3 + \dot{\imath}(A1) + \dot{\imath}(A1)\left[2 + \tfrac{1}{2}\dot{\imath}(A2)\right]$$
$$= 3 + 3\dot{\imath}(A1) + \tfrac{1}{2}\dot{\imath}(A1)\,\dot{\imath}(A2)$$

The Temporary Set method for intersecting the sets would require the same search of path 1 and only a single complete search of path 2. The result would be two temporary sets of A-strings which must be searched again to perform the intersection. The minimal structure for the temporary sets must be an E-string defined over each of the sets of A-string instances. The paths needed to search this structure would be

$$\text{TA1} \longleftarrow \text{TE1} \longleftarrow \text{ENTRY} \qquad (3)$$

$$\text{TA2} \longleftarrow \text{TE2} \longleftarrow \text{ENTRY} \qquad (4)$$

Intersecting TA1 and TA2 using the Alternating Search method would take

$$1 + \dot{\imath}(TA1) + \dot{\imath}(TA1)\left[1 + \tfrac{1}{2}\dot{\imath}(TA2)\right] \qquad \text{pointer retirevals.}$$

Since $\dot{\imath}(TA1) = \dot{\imath}(A1)$ and $\dot{\imath}(TA2) = \dot{\imath}(A2)$, the total retrievals needed for intersection by the Temporary Set method is given by

$$3 + \dot{\imath}(A1) + 2 + \dot{\imath}(A2) + 1 + \dot{\imath}(A1) + \dot{\imath}(A1)\left[1 + \tfrac{1}{2}\dot{\imath}(A2)\right]$$
$$= 6 + 3\dot{\imath}(A1) + \dot{\imath}(A2) + \tfrac{1}{2}\dot{\imath}(A1)\,\dot{\imath}(A2)$$

The New String method inveloves the creation of a new linking string structure that embodies the desired set intersection. Obviously the intersection must be performed once during the formation of the links. The Alternating Search method is used and as each pair of A-string instances is retrieved a new linked set is formed on the path $A2 \leftarrow A1 \leftarrow NL1$ where NL1 is a new L-string that links an instance of A1 with the matching instance of A2. Presumably a higher level E-string will be proveded to give access to the new linked sets, that is to the instancesof the new L-string. Thus the new path will be $A2 \leftarrow A1 \leftarrow NL1 \leftarrow NE1 \leftarrow ENTRY$

To access a pair of A-string instances from this path will require 1 pointer retrieval for NE1 and 3 retrievals for each instance of NL1, for a total of $1 + 3\,\dot{\imath}(A1)$ retrievals. The whole intersection process, then, requires

$$3 + 3\,\dot{\imath}(A1) + \tfrac{1}{2}\,\dot{\imath}(A1)\,\dot{\imath}(A2) + 1 + 3\,\dot{\imath}(A1)$$

$$= 4 + 6\,\dot{\imath}(A1) + \tfrac{1}{2}\,\dot{\imath}(A1)\,\dot{\imath}(A2) \qquad \text{retrievals.}$$

In summary, letting the instance counts be $\dot{\imath}(A1) = 4$ and $\dot{\imath}(A2) = 10$, the values obtained for each of the three methods are

$$3 + 3\dot{\imath}(A1) + \tfrac{1}{2}\,\dot{\imath}(A1)\,\dot{\imath}(A2) = 35 \qquad \text{(Alternating Search)}$$

$$6 + 3\,\dot{\imath}(A1) + \dot{\imath}(A2) + \tfrac{1}{2}\,\dot{\imath}(A1)\dot{\imath}(A2) = 48 \qquad \text{(Temporary Set)}$$

$$4 + 6\,\dot{\imath}(A1) + \tfrac{1}{2}\,\dot{\imath}(A1)\,\dot{\imath}(A2) = 48 \qquad \text{(New String)}$$

Comparing the values we see that for this example the Alternating Search method will always be preferable, for any values of (A1) and (A2).

Generalizing the above discussion, consider the two paths

$$A1 \leftarrow S1 \leftarrow \ldots \leftarrow Sn \leftarrow ENTRY \qquad (1)$$

$$A2 \leftarrow T1 \leftarrow \ldots \leftarrow Tm \leftarrow ENTRY \qquad (2)$$

where the instances of A1 and A2 are the sets to be intersected. The cost of intersection measured in pointer retrievals will be:

Alternating Search method -

$$\ell(1) + F(A1) + c(A1)\left[\ell(2) + \tfrac{1}{2}\,i(A2)\right]$$
$$= \ell(1) + F(A1) + \ell(2)c(A1) + \tfrac{1}{2}c(A1)\,i(A2)$$

F(A1) is a function which gives the expected number of retrievals needed to find c(A1) A-string instances in a total collection of $i(A1)$. At least c(A1) instances will be accessed and at most $i(A1)$. It is assumed that the desired instances are distributed randomly within the linked set. F is defined as

$$F(A1) = \frac{\sum_{k=c(A1)}^{(A1)} k\left[\binom{k}{c(A1)} - \binom{k-1}{c(A1)}\right]}{\binom{(A1)}{c(A1)}} \qquad \text{for } i(A1) > c(A1)$$

$$= (A1) \qquad \text{for } i(A1) = c(A1)$$

Temporary Set method -

$$\ell(1) + F(A1) + \ell(2) + i(A2) + 1 + c(A1) + c(A1)\left[1 + \tfrac{1}{2}\,i(A2)\right]$$
$$= \ell(1) + F(A1) + \ell(2) + i(A2) + 1 + 2c(A1) + \tfrac{1}{2}c(A1)\,i(A2)$$

Note that while only c(A1) instances of A1 must be retrieved, all instances of A2 are needed because it is possible that all of them will be involved in a match.

New String method -

$$\ell(1) + F(A1) + \ell(2)c(A1) + \tfrac{1}{2}c(A1)\,i(A2) \qquad \text{(to form the links)}$$
$$+ 1 + 3c(A1) \qquad \text{(to search the new string structure)}$$

Comparing the general results for the three methods, several observations can be made.

First, it is obvious that for a single intersection, the New String Method must always require more retrievals than the Alternating Search Method since the former includes an activation of the latter. However, once the new link strings have been established, they are available for use in answering future requests. The link forming process is a one-time fixed cost in the New String method and if the intersection is performed many times, the cost per intersection may well be lower than for repetitive use of the Alternating Search method. The point of cost indifference between these two methods could be easily determined.

Second, the choice between the Alternating Search and Temporary Set methods will depend on the comparison of the values of the expressions

$$\ell(2)\ c\ (A1) \qquad \text{and} \qquad \ell(2) + i(A2) + 1 + 2c(A2)$$

It is clear that as c(A1) becomes larger, the Alternating Search process becomes less attractive since the entire second path will have to be searched more times. Similarly, the longer path 2 is, the more favorable the Temporary Set method appears. This comparison can be made dynamically to choose the most efficient method.

From the above we see that the choice of intersection procedure may depend on the expected frequency of occurrence of certain queries, perhaps derived from the history of user requests. Also, factors such as the cost of storage and catalog search time may be important. Since these areas have not yet been explored, we will not attempt at this time to give formal rules for translation of RIL statements into RDL procedures. The preceding discussions can, however, provide guidelines for this translation process and suggest directions for further development.