The system architecture of EAS-E: An integrated programming and data base language

by D. P. Pazel
A. Malhotra
H. M. Markowitz

EAS-E is an application development system based on an entity-attribute-set view of system description. It consists of a procedural language for manipulating data base and main storage entities, and direct (nonprocedural) facilities for interrogating and updating data base entities. The EAS-E software itself was implemented with the entity-attribute-set view. This paper reviews some of the EAS-E features and considers some of its implementation details. This paper is both an introduction to the EAS-E software architecture and an example of the usefulness of the entity-attribute-set view.

EAS-E (pronounced "easy") is an application-developed system based on an entity-attribute-set view of system description. EAS-E allows the application developer to manipulate higher-level data structures in main storage and in the data base with equal facility. In particular, it allows him to work with entities, attributes, and sets in main storage and in the data base as easily as he works with main storage variables in conventional programming languages.

The application designer conceives the application in terms of the *entities* (objects and things) that must be remembered, their *attributes* (properties), and the *sets* (order collections of entities) that they own or belong to. The application can then be implemented in EAS-E, which directly supports operations on entities, attributes, and sets.

EAS-E consists of a procedural language for manipulating data base and main storage entities, and direct (nonprocedural) facilities for interrogating and updating the data base entities. The virtues of the EAS-E entity-attribute-set view with respect to building application systems are presented in References 1 and 2. Reference 1 also compares programs written in EAS-E with programs written in other programming systems. In the present paper we discuss the EAS-E software and the advantages of the entity-attribute-set view in building system software.

Simple EAS-E commands written by an application programmer, e.g., to CREATE a data base entity, FILE it into a set, or FIND it later, can result in quite

complex actions. The consequences of such actions, although transparent to the user, are to perform such actions as communicating information to and from the data base, efficiently manipulating very large sets, and ensuring the integrity of the data base. Thus the programmer communicates his intentions in a compact, readable form and the EAS-E software translates those intentions into the detailed computer actions required to execute them efficiently.

Very little is required to go from the conceptual model to the application program.

In the first part of this paper, the general character of the EAS-E language is explored. A simple example is first modeled in the EAS-E philosophy or world view. This same example is then presented as a simple main storage program written in the EAS-E language. Finally, the example is seen again as a data base application written in EAS-E. Very little is required to go from the conceptual model to the application program. The paper then shows the data base query and modification aspects of EAS-E. The concluding section shows the modeling of the execution environment of the EAS-E application system using the very philosophy on which EAS-E is based. Using this, structure and actions required by the EAS-E execution environment become clear. Thus, despite the complexity of these actions, the EAS-E software was implemented in a relatively short time by the authors with a relatively small amount of source code. We believe that the large amount of function produced per unit of resource expended was due in large part to the use of the entity-attribute-set view in the building of the EAS-E software itself. In summary, the following are highlights of EAS-E:

- It is an integrated data base and programming language that simplifies the specification of complex data base actions.
- It incorporates a powerful modeling philosophy.
- EAS-E is easy to use.

The EAS-E world view

The primitive concepts that comprise the world view behind our integrated programming and data base language are Entity, Attribute, Set, and Event (thus EAS-E). The power of these concepts has been used previously, for example, in two widely used simulation languages, SIMSCRIPT[3,4] and GASP.[5,6]

An *entity* may be thought of as a distinct structure or object. If one were to model a governmental system, for example, one might consider states and cities as *entity types*. When we refer to an entity type, we refer to a class of similar entities. For example, in a model containing 99 cities, CITY is the entity type of which there are 99 instances or entities. STATE is another entity type with one or more instances.

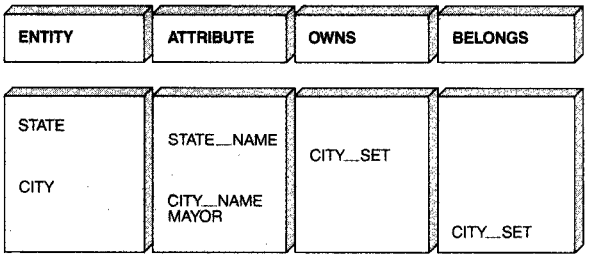
An *attribute* is a property or characteristic of an entity. Each attribute takes on values from some domain, e.g., real numbers, integers, alphanumeric strings, and pointers to user-defined entities. An attribute has at most one value at any time, or it may be undefined. For example, the entity type CITY may have CITY_NAME and MAYOR as attributes, and the entity type STATE may have STATE_NAME as an attribute.

A *set* is a collection of zero, one, or more entities of one or more entity types; a set is owned by an entity. For example, in the system with entity types CITY and STATE, we may introduce the set called CITY_SET owned by entities of type STATE, with member entities of type CITY. In the representation of the system, each STATE owns a CITY_SET consisting of some number of CITY entities.

These entity, attribute, and set facilities can be used to implement more complex structures such as stacks, pipelines, trees, and bills of material in a straightforward manner, as discussed in Reference 3. There are no limits on the number of sets an entity type may own or belong to. Also, there is flexibility in allowing entities of various types to belong to the same set or to own the same set. With this flexibility, the application modeler has full power in modeling structures as complex as he could possibly wish.

A visual aid for expressing the relationships among entities, attributes, and sets is illustrated in Figure

Figure 1 An entity-attribute-set (EAS) description of a system



1. The entity types are listed with their attributes, the sets they own, and the sets to which they belong. The information in Figure 1 is referred to as the entity-attribute-set (EAS) description of a system.

In working with an EAS structure, there are five basic actions out of which higher-level actions may be built. One may CREATE an entity, that is, make an instance of an entity type. One may assign values to attributes of entities. Entities may be FILEd in or REMOVEd from sets. Finally, entities may be DESTROYed; that is, an instance of an entity of some type may be annihilated. To add a new city to the system, for example, one CREATEs a CITY entity, assigns values to its CITY_NAME and MAYOR, and FILEs it into the CITY_SET of STATE.

The programming and data base language

EAS-E has been implemented on VM/370 at the IBM Thomas J. Watson Research Center, where it supports several applications. In this section, we first present EAS-E as a programming language. All basic operations are defined for private work spaces in main storage. We then present EAS-E as a data base language in which the basic operations extend naturally to data base concepts. Finally, we discuss the facilities that EAS-E provides to change data base definitions and modify existing entities to conform to new definitions. In the following sections, we discuss the implementation of these facilities.

EAS-E as a programming language. EAS-E includes an English-like procedural programming language that embodies many such standard language features as various data types, input/output, and control statements. EAS-E also features simple methods of defining and manipulating main storage EAS structures. Such structures are defined by means of the EVERY and DEFINE statements. The information in Figure 1, for example, is written as follows:

```
EVERY STATE HAS A STATE_NAME, AND OWNS A CITY_SET
EVERY CITY HAS A CITY_NAME, A MAYOR,
          AND BELONGS TO A CITY_SET
DEFINE STATE_NAME, CITY_NAME, AND MAYOR AS TEXT VARIABLES
```

Sets in EAS-E may be ordered in any one of the following three ways: (1) First In First Out (FIFO), (2) Last In First Out (LIFO) and (3) RANKED, that is, sorted according to the values of one or more attributes of the member entities. In main storage, these three types of sets are implemented as linked lists. In the given example, CITY_SET may be defined as FIFO or LIFO. Alternatively, CITY_SET can be ranked by CITY_NAME to provide an alphabetical ordering of the cities, specified as follows:

```
DEFINE CITY_SET AS A SET RANKED BY CITY_NAME
```

Entity and set definitions are contained in the initial section of an EAS-E program, called the PREAMBLE. This is followed by the executable source code. As mentioned earlier in this paper, EAS-E has many of the standard features of programming languages. We shall not discuss these here; instead we concentrate on the statements that manipulate EAS structures special to EAS-E. Statements corresponding to the five basic actions listed in the previous section are the following: CREATE an entity, DESTROY an entity, assign values to attributes with LET or READ statements, FILE entities into sets, and REMOVE entities from sets. The following example illustrates how these statements may be combined to add a city to a state:

```
CREATE A CITY
LET CITY_NAME=|GREENVILLE|
LET MAYOR=|JEAN GREEN|
FILE CITY IN CITY_SET(STATE)
```

EAS-E also provides a simple syntax for finding specific entities that are based on given attribute values. For example, if one wants to find the CITY with a CITY_NAME of GREENVILLE in the CITY_SET owned by STATE, and if found remove it from the set and delete it from the system, he writes the following procedure:

```
FIND THE CITY IN CITY_SET(STATE) WITH CITY_NAME=|GREENVILLE|
IF FOUND
  REMOVE CITY FROM CITY_SET(STATE)
  DESTROY CITY
ELSE
```

EAS-E as a data base language. In the development of an EAS-E data base application, the user has a

particular overview. Entity, attribute, and set structures are stored in the data base. The data base resides in a separate virtual machine and is overseen by a *custodian* program. The custodian manages the data and can respond to simultaneous requests from several virtual machines that are running EAS-E programs.

To define data base entity types, the user prepares a file of definitions similar to that of a PREAMBLE and communicates them to the data base. Then the user writes a series of programs to work with the data base, manipulating the previously defined structures. These programs are then compiled and executed.

To write a program that works with the data base, the user must specify which entity types are to be maniplated.

If the entities and sets of the preceding section were to define a new data base system called GOVERNMENT, the user would begin by transmitting the following file of definitions to the custodian:

```
DATA BASE DEFINITIONS FOR DATA BASE GOVERNMENT

EVERY STATE HAS A STATE_NAME AND OWNS A CITY_SET
DEFINE STATE_NAME AS A TEXT VARIABLE

EVERY CITY HAS A CITY_NAME, A MAYOR,
        AND BELONGS TO A CITY_SET
DEFINE CITY_NAME, MAYOR AS TEXT VARIABLES

DEFINE CITY_SET AS A SET RANKED BY CITY_NAME

END
```

After these definitions have been stored in the data base, the user is ready to work with them, creating, destroying, filing, and removing entities, and setting attribute values. This can be done either with the EAS-E procedural language discussed in this paper, or with the direct (nonprocedural) facilities. Procedural commands for doing this are essentially the same as those for main storage entities, but with a few differences.

To write a program that works with the data base, the user must specify which entity types are to be manipulated. This is done in the PREAMBLE of the program with the DATA BASE ENTITIES INCLUDE statement. For example, to work with the entity types STATE and CITY, the user writes the following statement:

```
DATA BASE ENTITIES INCLUDE STATE AND CITY FROM GOVERNMENT
```

Attributes and sets of STATE and CITY need not be specified in the PREAMBLE. The compiler obtains this information from the definitions stored in the data base.

The user works with data base entities much as though working with main storage entities. For example, to loop over all data base entities of a given type, such as CITY, he may write the following statement:

```
FOR EACH CITY, DO...
```

With that, the executing program acquires each CITY entity from the data base, one at a time. Alternatively, the FIND statement can be used to find a particular entity of a given type whose attributes satisfy certain criteria. This and some of the previous concepts are used in the following example of a program to add the city GREENVILLE to the state OHIO:

```
PREAMBLE
NORMALLY ACCESS IS READ.WRITE
DATA BASE ENTITIES INCLUDE STATE AND CITY FROM GOVERNMENT
END
FIND THE STATE WITH STATE_NAME=|OHIO|
CREATE A CITY
LET CITY_NAME=|GREENVILLE|
LET MAYOR=|JEAN GREEN|
FILE CITY IN CITY_SET(STATE)

END
```

Unlike main storage entities, data base entities are addressed by variables of data type REFERENCE. Reference variables may be declared anywhere in a program with an access mode of read.write or read.only. A reference variable with the prevailing access mode is automatically provided for each entity type specified in the DATA BASE ENTITIES INCLUDE statement.

Each data base entity has a unique data base *identification number*. In the current implementation, the identification number consists of three integer values, namely a *type* number that identifies the entity type, a *slot* number that serializes the instance of this type, and a *dash* number that indicates the number of times this slot has been occupied. Identification numbers are stored in variables of type IDENTIFIER. They are helpful in accessing entities explicitly. For example, if ID is an identifier variable and REF is a reference variable, the assignment

```
LET REF=ID
```

results in bringing from the data base the entity whose identification number is given in ID. The entity is brought read.only or read.write, depending on the access mode of the reference variable.

Simple queries can be written as small EAS-E programs.

When a program ends normally (i.e., not by crashing) the changes made to the data base are committed (made permanent). In addition, changes made up to a particular point may be committed by execution of the following statement:

```
RECORD ALL DATA BASE ENTITIES
```

If the user wishes to undo the data base manipulations prior to a RECORD, the UNLOCK ALL DATA BASE ENTITIES statement may be used. In case of system crashes, either the entire contents of an implicit or explicit record will be reflected by the data base or none of its contents will be reflected.

Queries in EAS-E. Simple queries can be written as small EAS-E programs. For example, the following program prints the names and mayors of all the cities in New York:

```
PREAMBLE
DATA BASE ENTITIES INCLUDE STATE AND CITY FROM GOVERNMENT
END
FIND THE STATE WITH STATE_NAME=|NEW YORK|
FOR EACH CITY IN CITY_SET(STATE)
PRINT 1 LINE THUS...
             CITY                          MAYOR
PRINT 1 LINE WITH CITY_NAME(CITY) AND MAYOR(CITY) THUS...
      **********************      ************************

END
```

Here the PRINT statement prints the line(s) following it exactly as specified, except that the asterisks are replaced by variable names.

Such a program is easy to write and allows a common query to be run over and over again. A more sophisticated version of the example would parameterize the program on the state name and perhaps the information to be displayed. In fact, EAS-E users tend to write query generators tailored to the queries they need most frequently.

For unusual queries and for looking through the data base, BROWSER, the full-screen nonprocedural facility mentioned earlier, is the most convenient. BROWSER allows one to move through the data base mostly by pressing program function keys on the terminal; it also allows the specification of simple reports with headings, totals, etc. BROWSER is described in References 2 and 7.

Modifying data base definitions. At any time after the GOVERNMENT data base is first defined, the definitions stored in it may be modified by statements such as those in the following program:

```
DATA BASE DEFINITIONS FOR DATA BASE GOVERNMENT

MODIFIED DEFINITION
EVERY STATE HAS A STATE_NAME, OWNS A CITY_SET
           AND A COUNTY_SET
DEFINE STATE_NAME AS A TEXT VARIABLE

NEW DEFINITIONS
EVERY COUNTY HAS A COUNTY_NAME, A COUNTY_SEAT,
      AND BELONGS TO A COUNTY_SET
DEFINE COUNTY_NAME, COUNTY_SEAT AS TEXT VARIABLES
DEFINE COUNTY_SET AS A SET RANKED BY COUNTY_NAME

END
```

Since the definition for CITY has not changed, it need not be resubmitted.

At this point, there are both an old and a new definition for STATE. Existing entities of type STATE are in the format defined by the old definition. In transforming the individual entities to the format defined by the new definition, they pass through another format called a dual format. The dual

format is a combination of the old and the new formats; i.e., it consists of two distinct entities, one in the old and the other in the new format. Thus each individual STATE can be in one of the following three formats: (1) old format, (2) dual format, or (3) new format. Entities are converted from old to dual format when they are first pointed to by variables of data type DUAL REFERENCE. This consists of the entity in the old format followed by an empty new format. One can now copy the common attributes and sets from the old into the new format with the following statement:

```
MOVE THE COMMON ATTRIBUTES AND SETS OF STATE
```

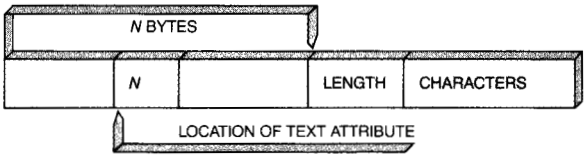
One may then fill in the other attributes of the entity in the new format and populate the new sets that it owns. In doing so, one can refer to attributes and sets of the old or new entity by prefixing their names with O_ and N_, respectively. When no longer needed, the old version of the entity may be destroyed, putting the entity into new format. When all the entities of a given type are in the new format, the custodian permits the old definition to be purged.

EAS-E implementation structures

Both the EAS-E compiler and data base custodian have been designed and built using the EAS philosophy. This has reduced several-fold the time required to implement the language and data base management system. The EAS view also pervades the background environment for an executing program. This environment is the principal topic of the remainder of this paper.

Data base entity structure. Data base entities are maintained on permanent storage and managed by the custodian. When a program requests an entity, the custodian fetches the entity from permanent storage and sends it to the program's main storage. At this point, the data base entity consists of a contiguous piece of storage with attributes laid out sequentially. The compiler uses the definition of the entity type to compute the offsets of these attributes. The offsets occur in fixed positions for all attributes of a given type. In the case of such variable-length attributes as TEXT, which may have arbitrary length, the offset points to a field that contains a relative displacement value. Figure 2 illustrates the way in which this displacement indicates where in the entity structure the text and its length are embedded.

Figure 2 A data base entity with a text attribute



Another consideration in the layout of a data base entity concerns ranked set structures. Data base ranked sets are implemented in a manner quite different from main storage ranked sets. In order to provide rapid access to members in large sets, some information about the ranking attributes of the member entities is embedded in the entity structure of the owner.[1] Since the amount of ranking information is variable, the displacement of the ranking information, like the displacement of TEXT attributes, is stored in a position indicated by a fixed offset.

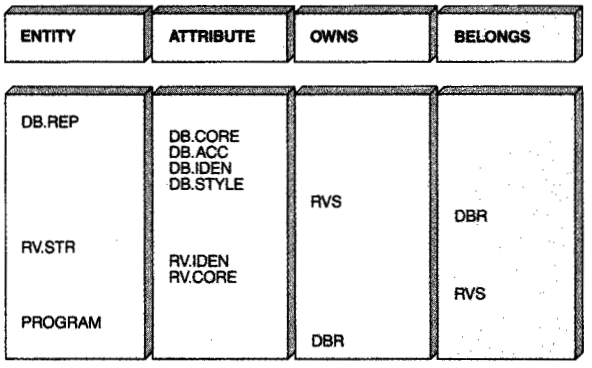
To summarize, a data base entity starts with fixed-length attributes and the displacements of variable-length attributes stored in positions that are constant for the entity type. This is followed by the variable-length attributes.

Fundamental system structures. To enable a user to work with data base entities in a manner equivalent to main storage entities, EAS-E provides mechanisms that (1) bring data base entities into main storage; (2) keep track of what has been brought into main storage and where it is located; and (3) record information in the data base. The structures used for this are now described.

Every EAS-E executing program is automatically provided with a main storage entity called the PROGRAM. Attributes assigned to the PROGRAM are in essence global variables. Sets owned by the PROGRAM are also global or *universal* sets.

A *data base representative*, or DB.REP, is a type of main storage entity used to keep track of data base entities brought into main storage. Each data base entity brought into the environment of an executing EAS-E program has a unique DB.REP. Essential attributes of DB.REP are the pointer to the main storage representation of the entity (DB.CORE), the access mode (DB.ACC), and the unique identifica-

Figure 3 An EAS description of EAS-E system structures



tion number for that entity (DB.IDEN). The PROGRAM owns a FIFO set called DBR to which the DB.REPs belong.

As mentioned earlier, reference variables are used to refer to data base entities. In fact, the reference variable does more than merely point to the main storage location of a data base entity. It encapsulates all the information about the existence of the data base entity in main storage. Thus each reference variable is a pointer to a main storage entity called a *reference variable structure* or RV.STR, which has as attributes the identification number (RV.IDEN), and a pointer to the main storage representation of the data base entity (RV.CORE). Also, since more than one reference variable can point to a data base entity, the RV.STR belongs to a set called RVS which is owned by the DB.REP entity.

The EAS structure of these relationships is shown in Figure 3. The entity-attribute-set names shown are not the actual names used in our implementation. The names used here provide readability. Our implementation is designed to conform to conventions that distinguish system-defined and user-defined names; it is discussed in Reference 8. Later in this paper, we show that DB.REP and RV.STR have additional attributes. These may be ignored for the moment until we discuss the structures needed for the modification of data base definitions.

As an example of how these structures are used at execution time, consider the actions associated with the statement CREATE A CITY. First, a request is made to the data base custodian for a unique identification number for the new entity. Using information provided in the object code, the executing program builds an empty main storage version of the entity. A DB.REP is created for the entity, its attributes are filled in, and it is filed into the DBR set. Next, an RV.STR is created, its attributes are filled in, and it is filed into the RVS set of the DB.REP. The reference variable CITY is now made to point to the RV.STR.

As another example, consider the actions associated with a statement that brings an entity from the data base, for example, by setting a reference variable to an identifier variable. The PROGRAM's DBR set is searched for an entity with that identification number. If the entity is not found there, it is requested from the custodian. Then a DB.REP is created, its attributes filled in, and it is filed in DBR. If the entity is found in the DBR set, its access mode is checked and upgraded from read.only to read.write if necessary. If the access mode is upgraded, notification is sent to the custodian. Finally, the attributes of the RV.STR to which the reference variable will point are filled in, and the RV.STR is filed into the RVS set of the DB.REP.

Data base set organization and manipulation. As previously mentioned, the EAS-E language provides three kinds of set organizations—First In First Out (FIFO), Last In First Out (LIFO), and RANKED organizations. LIFO and FIFO data base sets are held together as linked lists (like main storage sets). First- and last-member attributes are automatically defined for the owner entity, and successor and predecessor attributes are defined for any member entity. These member attributes point to the first and last members of the set and the successor and predecessor in the set of the particular member, respectively. The owner entity may also keep a membership count attribute, and the member entity may have an attribute pointing to the owner. For FIFO and LIFO sets, the difference between main storage sets and data base sets is that set pointer attributes are IDENTIFIER variables.

Ranked data base sets, on the other hand, are based on an entirely different organization. If ranked data base sets were implemented as linked lists, a search for a member with given attribute values would proceed by considering each member in turn until the desired one was found. Since accessing many members of a large data base set can be very time-consuming, such an implementation is unreasonable. To overcome this problem, ranked data base sets are organized as balanced trees.[9] A bal-

anced tree has a root node associated with the owner entity from which a tree of subnodes extends, with the leaf nodes pointing to the member entities. Information is maintained at each node that relates to the value ranges of the ranking attributes on the offspring nodes. The fanout at each node is quite large and is based upon keeping the size of the node entity close to one page. The implementation of ranked data base sets is discussed in more detail in Reference 1.

The implementation of ranked data base sets requires that the structure of data base entities be somewhat different from main storage entities, where ranked sets are implemented as linked lists. For example, the top node of a ranked set is included in the structure of the owner. In order for the system routines to locate that information, EAS-E generates an attribute in the entity that contains a displacement to the ranked set information relative to the beginning of the owner. Also, the tree nodes require the existence in the data base of an automatically defined entity type. The EAS-E system manipulates these entities according to the needs of the balanced trees. The extra entity attributes and the data base node entities are transparent to the user.

Looping through data base sets. The compilation of a loop statement in EAS-E, such as

```
FOR EACH CITY IN CITY_SET(STATE)
```

results in a simple loop control when the set is a linked list. More precisely, the *first* or *last* pointer of the owner is used to initiate the loop, and the *successor* or *predecessor* pointers of the members are used to continue the loop until the last member is processed. If selection clauses such as WITH CITY_NAME=|GREENVILLE| are appended to the previously looping statement, a series of tests (IF statements) are generated within the loop proper. In this case, although the domain of the loop appears smaller, the full set must still be searched. That is, each member of the set must be brought into main storage and tested.

The ranked set organization just described is designed to help locate entities that meet given conditions, without bringing in each member of the set. To do this, the compiler must identify and extract the bounds given on the loop constraints in the source program and arrange to pass these restrictions at execution time to a set-scanning mechanism.

To identify the selection criteria, the compiler scans the selection clauses on a loop statement for constraints of the following form:

```
< RANKING ATTRIBUTE > < LOGICAL RELATION > < EXPRESSION >
```

Here EXPRESSION is an arithmetic expression that does not contain any ranking attributes. We call such a constraint a P-CONSTRAINT. For each ranked-set loop, the compiler examines the appended constraints and seeks out the first *n* P-CONSTRAINTs that are related to the ranking attributes in order and are linked by "and"s, for maximum *n*; that is,

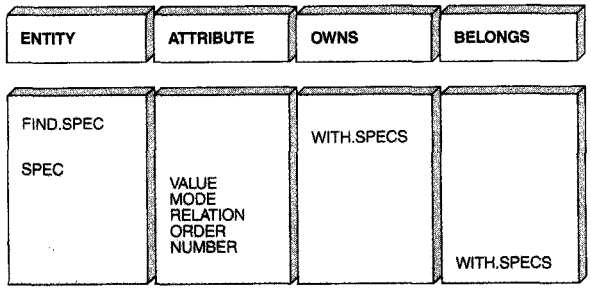
```
< P-CONSTRAINT > < "AND" > < P-CONSTRAINT > < "AND" > .....
```

The problem of passing the selection information at execution time to the loop-searching mechanism has been addressed by designing an EAS structure to contain that information.

Here the P-CONSTRAINTS are ordered by the ranking attribute. These P-CONSTRAINTS are then translated into object code that generates a selection structure that imposes these limits on the domain of the loop. The selection structure is discussed more fully later in this paper. Constraints that do not fit into this pattern are translated into a set of conditional clauses in the main body of the loop, as usual. Thus, members are selected in accordance with the P-CONSTRAINTS that embody the specifications on the ranking attributes; if necessary, they are brought into main storage and tested for other criteria.

The problem of passing the selection information at execution time to the loop-searching mechanism has been addressed by designing an EAS structure to contain that information. Calls are then generated in the loop structure that fill in those structures and

Figure 4 EAS view of selection structures



eventually pass them to the loop mechanism at execution time. An EAS view of a selection structure is shown in Figure 4.

The selection structure consists of an owner node (FIND.SPEC) that owns a set of SPECS, each of whose members contain all the information about a P-CONSTRAINT. VALUE is the execution-time value of the EXPRESSION portion of the P-CONSTRAINT. MODE indicates the data type of the ranking attribute being constrained. RELATION specifies how the ranking attribute relates to the VALUE, e.g., $=$, $<$, $>$, etc. ORDER indicates whether ranking on that attribute is by high or low value. NUMBER is the ordinal number of the ranking attribute, that is, first, second, third, and so forth. This structure is used by the looping mechanism to search the balanced tree of the set. Selection structures are also used in FILE and REMOVE operations to find where a new member should go and to find the member that is to be removed. The creation, filling in, utilization, and clean-up of these structures is completely transparent to the user.

RECORD and UNLOCK. The execution of the statement RECORD ALL DATA BASE ENTITIES results in making all creates, destroys, and other alterations of data base entities permanent to the data base. A RECORD may be issued at any time during program execution and may be specified with a HOLD option, in which case all read.write entities that have been accessed remain accessed upon completion of the RECORD. This saves communication overhead if the entities are used again. If HOLD is not specified, all read.write entities not pointed to by a reference variable (i.e., not being used) are released. This saves main storage space in the program. The following paragraph shows how the DB.REP and RV.STR structures are used in the RECORD operation.

The identification number of each read.write entity, as represented by DB.REPs in the DBR set, is written onto a communication buffer. If the entity has been destroyed, this fact is noted. Otherwise, the entity is rebuilt as a contiguous piece of storage, with text attribute and ranked set information appended to the entity and copied into the communication buffer. This buffer is transmitted to the custodian, who commits these changes to the data base. When control is returned from the custodian, the DBR set must be scanned once more to find the DB.REPs of entities recorded, to free the main storage version of those entities, to null out and remove RV.STRs from RVS sets, and finally to destroy their DB.REPs.

The effect of executing an UNLOCK statement is to release all data base entities. With this statement, any changes made in an executing program since the last RECORD are rescinded. This is, in effect, an undoing of data base actions before they are committed. The actions for this statement for each DB.REP in the DBR set are as follows: the main storage for the data base entity is released, the RV.STRs are nulled and removed from RVS sets, and the DB.REP is destroyed. The custodian is told that all entities this user has accessed are to be released.

Looping and auto-unlock. EAS-E provides the capability of looping over all data base entities of a particular type, as in executing FOR EACH CITY . . . . This is accomplished through a get-next operation, based on a sequential catalog of entities of a given type maintained by the custodian. To obtain the first entity, the get-next operation transmits a special identification to the custodian. On successive passes, get-next communicates the identification number of the previously obtained entity to the custodian and receives the next entity from the custodian, or it receives an indication of loop termination.

In looping over a large number of entities—as in FOR EACH CITY . . .—main storage can all be used up unless unneeded entities are released. To relieve the programmer from having to do this, EAS-E unlocks read-only entities that are not being used —that is, they have no reference variables pointing to them—whenever there are 30 such entities.

Modifying data base definitions. As previously noted, sometimes both an old and a new definition may exist for any entity type. An individual entity of a type may then be in one of three formats—old,

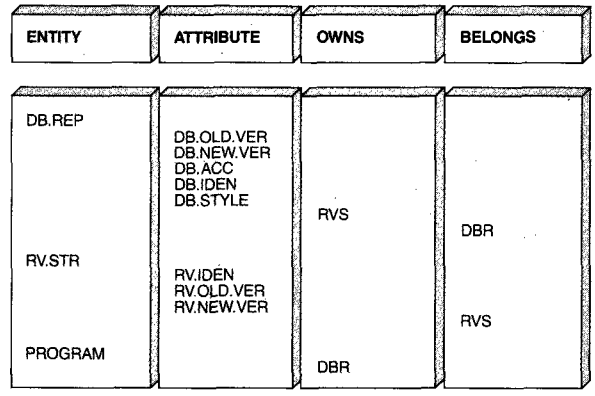
new, or dual. Because of this, the DB.REP and RV.STR are more complex than shown in Figure 3. Figure 5 provides a more complete description of EAS-E modification structures. (DB.CORE and RV.CORE in Figure 3 are referred to as DB.NEW.VER and RV.NEW.VER, respectively.) Since an entity can be in old, new, or dual format, each DB.REP and RV.STR has a pointer to the old version of the entity in main storage (DB.OLD.VER or RV.OLD.VER) and a pointer to the new version of the entity in main storage (DB.NEW.VER or RV.NEW.VER). Also, DB.STYLE indicates the current format of the entity.

When an entity that has two definitions is requested from the custodian, the following actions are performed. The custodian provides a code to the executing program indicating the format of the entity received from the data base. If the entity is in old format and is being referenced by a DUAL REFERENCE variable, it is put into dual format with the creation of a blank new version of the entity and by setting the DB.REP and RV.STR main storage pointers to the respective versions. If the entity is already in dual format, the data received from the custodian consist of the concatenation of the two versions of the entity. The system code then sets the DB.REP and RV.STR pointers to the respective entities. If the entity is in new format, then only DB.NEW.VER and RV.NEW.VER point to the received data.

EAS-E has been designed to accommodate data bases of arbitrary size, from very small to very large.

Depending on the format of the entity, different ways of packaging it at RECORD time are selected. In particular, if the entity is in dual format, both versions of the entity must be reassembled separately in a manner specified in the section on RECORD and UNLOCK. Then both entities are moved to contiguous storage in the communication buffer with the old version preceding the new version.

Figure 5 Description of EAS-E modification structures



Concluding remarks

This paper has presented a brief overview of the EAS-E modeling philosophy or world view and the EAS-E programming language. This approach to application development focuses sharply on clarity in data structures (i.e., entities-attributes-sets) and actions (i.e., events). We have shown that by using a programming and data base language based upon these elements, there is little effort in going from a conceptual model of an application to the application program itself. EAS-E has been designed to accommodate data bases of arbitrary size, from very small to very large.

The EAS-E modeling philosophy has been used throughout the design and implementation of the EAS-E application development system. The EAS-E compiler, custodian, and library routines are all written in the EAS-E language.

Areas for further research include such capabilities as the following: building of data base utility routines to be used by host languages, communication with several EAS-E data bases simultaneously, and single-user EAS-E data bases. In the latter case, the data would exist as a user's private data base, as opposed to residing in a separate service machine.

Cited references

1. A. Malhotra, H. M. Markowitz, and D. P. Pazel, *EAS-E: An Integrated Approach to Application Development*, Research

Report RC 8457, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598; also submitted to the *ACM Transactions on Database Systems*.
2. H. M. Markowitz, A. Malhotra, and D. P. Pazel, "The ER and EAS formalisms for system modeling, and the EAS-E language," *Proceedings of the Second International Conference on Entity-Relationship Approach*, Washington, DC, October 12–14, 1981, pp. 29–48; also, Research Report RC 8802, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.
3. H. M. Markowitz, "SIMSCRIPT," *Encyclopedia of Computer Science and Technology*, Vol. 13, J. Belzer, A. G. Holtzman, and A. Kent, Editors, Marcel Dekker, New York (1979), pp. 79–136; also Research Report RC 6811, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.
4. H. M. Markowitz, B. Hausner, and H. W. Karr, *A Simulation Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1963).
5. P. J. Kiviat, *GASP—A General Activity Simulation Program*, U. S. Steel Corporation, Applied Research Laboratory, Monroeville, PA (July 1963).
6. A. A. B. Pritsker, "GASP," *Encyclopedia of Computer Science and Technology*, Vol. 8, J. Belzer, A. G. Holtzman, and A. Kent, Editors, Marcel Dekker, New York (1977), pp. 408–430.
7. H. M. Markowitz, A. Malhotra, and D. P. Pazel, *The EAS-E Application Development Summary: Principles and Language Summary*, Research Report RC 9910, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.
8. A. Malhotra, H. M. Markowitz, and D. P. Pazel, *The EAS-E Programming Language*, Research Report RC 8935, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.
9. R. Bayer and K. Unterauer, "Prefix B-trees," *ACM Transactions on Database Systems* **2**, No. 1, 97–137 (March 1977).

General references

M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, "System R: A relational approach to database management," *ACM Transactions on Database Systems* **2**, No. 1, 97–137 (June 1976).

CODASYL Data Base Task Group Report, available from the Association for Computing Machinery, Inc., 1133 Avenue of the Americas, New York, NY 10036.

C. J. Date, *An Introduction to Database Systems, Third Edition*, Addison-Wesley Publishing Company, Inc., Reading, MA (1981).

Information Management System/360, General Information Manual: Program Product 5734-XX6, GH20-0765 06061, IBM Corporation; available through IBM branch offices.

Ashok Malhotra *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Malhotra has been a research staff member in the Computer Sciences Department at the Research Center since 1975. His current research in improved application development facilities is a manifestation of a general interest in making computers more accessible to the nonspecialist, especially the manager. Dr. Malhotra received his Ph.D. from M.I.T. in management; he has several years experience as a management consultant. He is editor of the *Program Development Quarterly*, an internal IBM newsletter devoted to advances in program and application development technology.

Harry M. Markowitz *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Dr. Markowitz has been a research staff member in the Computer Sciences Department at the Thomas J. Watson Research Center since he joined IBM in 1974. During this period his principal interest has been the design and development of the EAS-E application development system. Previously, while at the RAND Corporation (1952–1959 and 1960–63), he developed techniques for inverting very large but sparse matrices which are now widely used. Also at RAND he designed and supervised the development of the SIMSCRIPT simulation programming language. Dr. Markowitz received his Ph.D. in economics from the University of Chicago. In his Ph.D. dissertation he developed the "portfolio theory," which is now regularly taught in finance departments of business schools.

Donald P. Pazel *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Pazel is a research staff member in the Computer Sciences Department at the Thomas J. Watson Research Center. He joined IBM in 1973 at Morris Plains, New Jersey, where he worked on the Safeguard project for the Federal Systems Division. Since joining the Research Center staff in 1975, Mr. Pazel has worked in the areas of operating systems, compilers, and data bases. Mr. Pazel graduated maxima cum laude from LaSalle College, Philadelphia, in 1972 with a B.A. in mathematics, and received an M.S. degree in mathematics from the University of Virginia in 1973. He is a member of the Mathematics Association of America and the Association for Computing Machinery.

Reprint Order No. G321-5190.