Basic Database Concepts in ADAMS (Advanced DAta Manipulations System) Language Interface for Process Service

John L. Pfaltz, Sang H. Son, James R. French,
Paul K. Baron, David J. Kirks, and Ratko Orlandic

IPC-TR-87-001
November 30, 1987

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 22903

This research was supported in part by JPL under contract #957721.

IPC Technical Report 87-1

Basic Database Concepts in ADAMS (Advanced DAta Manipulation System) Language Interface for Process Service

John L. Pfaltz
Sang H. Son
James R. French
Paul K. Baron
David J. Kirks
Ratko Orlandic

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract:

Every database implementation is eventually grounded in a number of primitive concepts which are fundamental to understanding the system. ADAMS is based on five primitive concepts: attribute, co-domain, element, map, and set, which are developed in this preliminary design report. We then show that more traditional database constructs, such as relations, networks, and/or arrays, can be expressed in terms of these primitives.

This research was supported in part by JPL Contract #957721

1. Basic Concepts

1.1. Introduction

ADAMS, the Advanced DAta Management System of the University of Virginia Institute for Parallel Computation, has been designed to function as clean interface between user processes (typically running in a parallel processor environment) and one, or more, persistant data bases. It is not meant to serve as a traditional database language. In particular, it does not provide any mechanism for (1) interactive terminal interface, (2) user specified retrieval, or (3) manipulation of retrieved data. We expect that most user processes will employ an established database language such as SQL, ORACLE, M204, or Ingress to provide these functions. Instead ADAMS will provide mechanisms for defining and naming persistant database structures on permanent storage media, and for accessing these structures.

Thus ADAMS becomes primarily a tool for implementing other database languages and for porting them to a variety of hardware configurations.

This report is a preliminary design specification. It is admittedly incomplete; it is a working document. It does, however, capture in a relatively coherent way the many ideas that its authors have developed over a six month period of weekly meetings and discussions.

Basically, ADAMS provides a systematic way of dealing with sets in a database. But it must identify three very different kinds of sets and provide three different linguistic mechanisms to deal with each kind.

First, there are abstract sets of "values", where a *value* denotes some bit string that will be interpreted according to a pre-defined convention. For example, the set of "integers", the set of "reals", or the set of strings of 5 or fewer characters. Many programming languages call these sets "data types". We will call them *co-domains,* and discuss them more thoroughly in sections 1.4 and 4.1.2.

The basic ADAMS *set* is a set of "elements", where *elements* are considered to be any of a large variety of objects in an *name space.* Many authors would call these elements (or objects)---data structures. ADAMS provides computational procedures to manipulate these sets; and the term *set* will only be applied to sets of these computational elements, or structures. In the ADAMS language the word "set" will always refer to one of these.

Finally, there are abstract sets (or classes) of ADAMS elements. Each ADAMS element, or object, is one representative of the set, or class, of *all possible* objects of that kind. For example, there may be in the database a specific instance of a "linked list", or a several instances of a relation with a particular schema. ADAMS must provide a language in which one can describe different classes of elements. In many ways, this linguistic mechanism constitutes the key to the language interface described in this report.

Specific *co_domains* and *classes* are defined "syntactically" within the ADAMS language. Specific *sets* (of elements) are defined by construction (by computational process).

The underlying premise of ADAMS is that there are only a few very basic underlying concepts in database representation. We assume that *all* database models can be expressed in terms of these basic concepts, and thus *all* databases can be implemented using this reduced set of database concepts. Moreover, we assume that if one knows how these few concepts have been represented, then one knows how the entire database has been implemented.

The primitive concepts are

a) element
b) set
c) co-domain
d) attribute
e) map

where each of these will be described in detail in the next sections. In the last sections we illustrate how more traditional database models can be expressed in terms of these ADAMS concepts.

1.2. Elements

An *element* is a primitive concept in ADAMS. An element has no *a priori* form or structure. The only essential characteristic of an element is that it is *uniquely identifiable.* We will give two analogies to expand on the "element" concept.

In formal set theory, one traditionally defines a set as a "collection" (this primitive definition is always a bit circular) of elements. It is illrelevant what these elements "really are"; all that is required is that we be able to identify them, say as *a*, *b*, or c, so that we may make statements such as S = { *a, b* } or $c \in X \cap Y$. These elements are certainly not the strings "a", "b", or "c" themselves. And it makes no difference whether we ever know what the symbols *a, b* or *c* denote. "Meaning" may, or may not, later be ascribed to them.

A different, more concrete, analogy comes from the area of object oriented programing. An "element" may be thought of as an *object.* An object is any section of memory or secondary storage. An object is identifiable; it has an *address.* Being able to identify an object, that is knowing its "address", tells us nothing about the object itself. It may be a chunk of executable code, a linked list, a single data value, or a text file.

The concepts of *element* and *object* are so nearly synomous that we frequently use them interchangeably. "Objects" may be regarded as concrete representations of abstract "elements". We have chosen to use the term "element" simply because the term "object" has so many existing connotations in the computer literature [BBK87, CoM84, SSE87]. Even so, most of those connotations are completely compatible with our usage.

All elements (or objects, if you prefer) in an ADAMS database must be *typed,* that is belong to a known *class.* Except for the three system defined classes,

set,
attribute, and
map

the class (or type) of an element has no pre-defined significance. The creator of an element can assign any class designator whatever (for example, "Q", "Budweiser", or "XX3") to the element. Any significance attached to a non-system defined class is completely the responsibility of the user. In section 2.1 we will discuss the way that classes are defined.

The ADAMS database only requires that

(1) every element in the system must be assigned some type or class at the time of its creation, and

(2) if the element is to function as either a set, map, or attribute (as described in the next sections), it must be so classified.

1.3. Sets

The fundamental database "structure" in ADAMS is the *set.* If the elements of the set were "records", then one would customarily call the set itself a "file".

Sets of elements are characterized by the following properties

(1) A set is *uniquely identifiable.* It is itself an element.

(2) All sets are finite. They are *unordered.*

(3) The elements of a set must all be of the same *class,* or type. If, for example, several elements of class "Q" were created, all, or some, of them could be combined as a set. Elements of class "Q" and class "XX3" can not belong to a common set.

(4) Sets themselves are *elements* of the class SET. More accurately, the class of a set is "set of <class>", where <class> denotes the class of its constituent elements.

(5) All of the standard set operations (except unary complement) may be performed on ADAMS sets. Specifically, we recognize

union
intersection
relative complement
creation
element insertion
element deletion
set membership test
test for emptiness
cardinality
looping over the set

and others.

1.4. Co-domains

The concept of a "data value" is purposely left undefined in ADAMS. Data values are strings of bits that other processes manipulate in various fashions. ADAMS will store, and retrieve, such data values on the request of external processes, so it needs to know enough about such data values to facilitate their storage and later retrieval, but no more.

A "co-domain" may be regarded as a set of data values. For example, the half open interval [0,10) consisting of all real values x, $0 \leq x < 10.0$, could be a co-domain.

More precisely, in ADAMS, a *co-domain* is any regular set that is definable in LEX (or a LEX-like language). A co-domain is a "nameable" set, but it can not itself be an element (in the ADAMS sense) and can not be generally manipulated in the manner of the sets in the preceding section. The only set operation that can be performed with a co-domain is that of set membership. Given a particular value, it can be determined whether it is a member of the co-domain (i.e. accepted) or not.

Generally, a co-domain can be regarded as the specification of the legal form of a kind of data value.

There are two predefined co-domains. They are:

$elmid	-- the set of all recognized element identifiers
$names	-- the set of all recognized external names.

The co-domain *$elmid* needs little more explanation. All ADAMS elements are uniquely identified, *$elmid* specifies the form of these identifiers.

We must develop some background concerning the co-domain *$names*. A *name* is an externally accessible identifier. All ADAMS elements are uniquely identifiable, but such identifiers are commonly various forms of "virtual addresses" that are not accessible outside of the particular database structure in which the element is embedded. It need not be externally known, provided there is some procedural path by which it can be accessed. But other ADAMS elements should be "externally known". Suppose an ADAMS set denotes a "file" of "records", then the "name" of that file should be accessible so that various processes can use it. A *name* provides this external access. It may be found (or looked up) in a "dictionary" of names.

All elements are nameable, but most elements are not in fact uniquely named. Co-domains are named. Classes are named.

We close by noting that the ADAMS "co-domain" concept is analogous to that of "domain" in many other database models. And they will be used in a similar manner, for example, the value of any attribute mapping must be an element of a co-domain. We choose to call the concept a "co-domain" (rather than "domain") for two reasons. First, it is mathematically more correct since an attribute maps from a set of elements (its domain) to a set of values (its co-domain). Secondly, we choose to restrict the concept to regular sets. The different terminology emphasizes this restriction.

1.5. Attributes and Maps

Attributes and maps are both functions, that is they are singled valued. The only distinction is the nature of their respective image sets. An *attribute* is a function mapping elements into a co-domain; that is, the image of an attribute is a value. In contrast, a *map* is a function mapping elements into a class of elements. Its image is another ADAMS element. (In DAPLEX [Shi81] these concepts are called *property* and *function* respectively.)

If one thinks in terms of traditional file processing, then one can regard a "file" as a set of "records" (elements). An attribute, say "EMP_NUMBER", which associates a particular employee number with each element (record) of the set is then a "field" operator. If, instead, one is accustomed to thinking in terms of the relational model then the "attribute" concept is more familiar. A "relation" is a set of "tuples" (elements) and an attribute again associates specific data values with the tuple. The major difference is that, in ADAMS, sets themselves may have attributes.

Let *a* denote an arbitrary attribute, and let *x* denote any element in the database on which *a* is defined, then the ADAMS expression *x.a* denotes some string in the codomain of *a.* (Note, every co-domain has a NULL value; so the expression *x.a* is well defined whether the attribute *a* has, or has not, been assigned for *x.)* The ADAMS expression

```
assign (x.a, string)
```

assigns *string* as the value of *x.a* provided, of course, that *string* is actually in the co-domain (i.e. satisfies the defining LEX expression). These ways of referencing co-domain values correspond to *getDomVal* and *putDomVal* in [CAD87].

We observe that many computational processes using ADAMS as their database interface do not manipulate strings. They typically manipulate numeric values, reals and integers. A more characteristic processor-ADAMS interface would look like:

```
X := decode(x.a);
assign (x.a, encode(Y));
```

where X and Y are variables of the process itself, and *decode* and *encode* are standard procedures which can be defined at the same time that the co-domain of a is defined.

Normally, the user defines and names attributes. But there are two pre-defined attributes. They are:

$$\text{elem\_id} : U \rightarrow \$\text{elmid}$$

$$\text{class\_of} : U \rightarrow \$\text{names}$$

Here U denotes the universe of all elements.

Note that attributes are a kind of element. Consequently, we may create sets of attributes and apply standard set operations to them. This capability is important in both the relational approach to databases and the handling of array data. By the same token, an attribute may itself have attributes. It is not clear if this generality is of any value or not.

As we noted at the beginning of this section, a *map* is a function which maps an element into a *class of elements.* Thus application of a map to an ADAMS element yields another ADAMS element instead of a co-domain value produced by an attribute function. Some find it easier to think of a map as an element pointer value.

Frequently, the class of elements which constitute the image of a map is some restriction of the class SET. This introduces the ability to construct one-to-many and many-to-many maps, which we explore more fully in section 4.4.

1.6. Concise Summary of Primitive Concepts

All of the fundamental properties of ADAMS database concepts are outlined here.

(1) Element

- a) uniquely identifiable, but not nameable unless a set, attribute, or map.
- b) must be typed with some class membership.
- c) pre-defined types are:

    1) set
    2) attribute
    3) map

- d) may be a member in any set.

(2) Set

- a) nameable.
- b) all elements must be of the same class, or type.
- c) can be involved in all standard set operations.

(3) Attribute

- a) function from an element to a co-domain.
- b) nameable.
- c) pre-defined attributes:

    1) elem_id : $U \rightarrow$ $elmid
    2) class_of : $U \rightarrow$ $names

(4) Co-Domain

- a) regular set.
- b) nameable, but not manipulable.
- c) pre-defined co-domains:

    1) $names
    2) $elmid

(5) Map

- a) function from an element to an element (often a set).
- b) nameable.

The following sections describe syntactic constructs that assume these properties.

2. ADAMS Syntax

ADAMS is not primarily concerned with processing, even though it must be capable of some minimal processing. It is primarily concerned with the storage and retrieval of "things" (character or bit strings). Consequently, the language we are developing emphasizes constructs which can be used to describe the structure of such "things". In the preceding sections we have described the primitive "things" which ADAMS understands. They are:

elements, which can participate in any set operation

values, for which only set membership is defined.

But it is inconvenient to repeatedly develop database applications in terms of very primitive concepts. It is slow and error prone. In the visualization and subsequent design of databases we all think in terms of various kinds of composite structures such as, *records* and *files*, *relations* and *relationships.* These are the kinds of higher level concepts that are primitive in most database design. ADAMS can bootstrap itself up so that the database designer does indeed see these as the basic concepts. But to do this we must develop several other linguistic and meta-linguistic concepts.

2.1. Class Description

In ADAMS, sets, elements, co-domains, maps, and attributes denote the kinds of things that can be manipulated. In a sense they represent the *noun classes* of the language. Thus a "class" is simply the name of a class of things. We may speak of the class "SET", that is, the collection of all objects which are sets. But we may also speak of a particular "set", that is an instance of the class. Specific instances will constitute the most common *nouns* in the ADAMS language.

For pedagogical purposes in this report, whenever a word denotes an entire class, it will be set in UPPER CASE. Whenever the word identifies a specific instance, it will be set in "lower case" characters. "Reserved" words have been emboldened for emphasis. The following three

ADAMS statements follow this convention

Q **is a** CLASS << >>

Q_SET **is a** SET << **of** Q **elements** >>

q **belongs to** Q_SET << >>

"Q" denotes a class (which is arbitrary). "Q_SET" denotes the class of all possible sets whose elements are of class "Q". "q" denotes a particular set of some such (possibly zero) elements.

ADAMS does not enforce this upper-lower case distinction, but we find it valuable when creating ADAMS descriptions.

2.1.1. Predicates

A *predicate* is any expression, such as f(S, t), that evaluates to true or false. For example,

$$(\forall\ s \in S)\ [\ s.\text{class\_of} = \text{SET}\ ]$$

$$t.\text{cardinality\_of} < 9$$

are predicates.

Predicates can also involve attribute values, as shown in the next example. Briefly, given $x \in X$, a class of elements on which $a \in$ ATTRIBUTE is defined, then **x.a** will denote a specific value in co-domain of a. It is an attribute evaluation. With this abbreviated introduction to attribute evaluation,

$$(\forall\ x \in X)(\forall\ y \in Y)\ [\ x.a < y.a\ ]$$

is a valid predicate. Throughout ADAMS we assume two important facts; that

1) the values of any co-domain can be ordered, and

2) a predicate expression denotes a decidable process.

The first assumption can be assumed, if necessary, as a byproduct of the binary encoding scheme. The second assumption follows largely as a consequence of our requiring co-domains to be regular sets.

2.1.2. Restriction

It is common, in natural discourse, to define a new class of objects by further restricting an existing class. For example:

FISH **is a** ANIMAL
    << **in which** ($\forall$ x $\in$ FISH) [x.swims_in_water] >>

That is the class of FISH is a subset of the class of ANIMALS which is further restricted by the requirement that any member, x, of the class must "swim_in_water". Predicate restrictions are denoted by the clause

in which <predicate_expr>.

The predicate must be true in order for the element to have class membership. Predicates can be composed with the usual boolean operators, *and, or,* and *not.*

The predicate expression

($\forall$ x $\in$ <set_class>) [x.class_of = <element_type>]

which restricts the <set_class> to a single specific class of elements, is so ubiquitious that we have condensed it to a single restriction clause

of <element_class> **elements**.

"In which" clauses restrict the kinds of elements that can belong to a class. Readily, if X is a restriction of Y, which is in turn a restriction of Z, then X "inherits" the properties of both Y and Z. To be a member of X, x must satisfy $predicate_X$ as well as $predicate_Y$ and $predicate_Z$.

2.1.3. Term Definition

While most of the database concepts of ADAMS are defined in terms of moderately complex constructs based on very primitive concepts, we very seldom use these complex constructions directly. Instead we use various synonyms for the constructs. Natural languages all employ the same mechanism. In ordinary discourse we easily use the word "fish" instead of the phrase "animals that swim in water".

ADAMS uses two basic language constructs to enlarge its vocabulary.

```
<class_name> is a <class>
                << in which <restrictions and/or qualifications> >>

<instance_name> belongs to <class>
                << <further restrictions/qualifications (if any)> >>
```

The first defines (or names) an abstract "class" of things. The second names a specific instance (thing or object) within a class. Notice that the *is a* construct used to define new classes is a general class definition operator. It is not a specific *inheritance* operator as in [ACO85] or [BuA86] even though the defined class does inherit the properties of <class>.

The <class_name> or <instance_name>, together with its corresponding definition is automatically entered into the users local dictionary. It is now part of the user's own database language. The name and its definition may be later exported to the group and/or system dictionaries to become part of a larger database group or system vernacular.

2.1.4. Association

Attributes are defined on elements. That is to say, some kinds (classes or types) of elements have some attributes defined on them. One or more attributes are associated with a class of elements by the *having* construct. Consider the following sequence of ADAMS statements.

```
Q_A is a SET
        << of ATTRIBUTE elements >>

q_attributes belongs to Q_A
        << consisting of { a1, a2, a3 } >>

Q is a CLASS
        << having q_attributes >>

Q_SET is a SET
        << of Q elements >>
```

The set *q_attributes* denotes a specific set of the specific attributes, *a1, a2,* and *a3,* (which have presumably been defined earlier). Thus this sequence has declared that any element in any set of class Q_SET will be of class Q and will have these three attributes { a1, a2, a3 } defined on them.

For a different and slightly more complex example consider the following sequence in which we have used some more traditional file processing terms. In spite of the different terminology, the underlying structure of the classes Q_SET and P_FILE, is very similar. And the terms Q_A and DATA_FIELDS are completely synonymous; they denote precisely the same class—sets of attributes.

```
DATA_FIELDS is a SET
        << of ATTRIBUTE elements >>

p_rec_data belongs to DATA_FIELDS
        << consisting of { p_nbr, name, age, dept, salary } >>

P_REC is a CLASS
        << having attributes = p_rec_data >>

P_FILE is a SET
        << of P_REC elements >>
           having attributes = { date_last_mod }
        >>
```

Any set of class *P_FILE,* we would probably call a *personnel file.* Elements of the file are of class *P_REC;* we would probably call them personnel *records.* The data fields of each *record* are *p_nbr*, *name*, *age*, *dept*, and *salary*. While these attributes (fields) are associated with the elements (records) of the set (file), there is also an attribute (field) associated with the set (file) as a whole; that is, *date_last_mod,* the date the file was last modified.

The *having* construct can only associate extant sets with a class. In the preceding example, the set *p_rec_data* was explicitly named before defining the class P_REC. The enumerated set of defined attributes *{ date_last_mod }* was implicitly created in the definition of the class P_FILE. The set *p_rec_data* (since it has a name) is independently modifiable. The unnamed set *{ date_last_mod }* is not.

Note that both the classes *P_REC* and *P_FILE* have used the designator *attributes* to be an additional *synonym* for their associated sets. The use of such a synonymous designator is completely optional.

2.1.5. Internal Class Definition, and the *only* if Clause

In the preceding section we illustrated the creation of a class *P_FILE*. First, the class *DATA_FIELDS* denoting any set of attributes was declared. Then a specific set *p_rec_data* was created. The class *P_REC* denoting a class of elements (or records) all of which have the associated attributes in *p_rec_data* was declared. And finally, the class *P_FILE* could be declared as any set of elements (records) belonging to *P_REC*. At best, such a definition mechanism is "tedious". At worst, this mechanism (1) is error-prone, (2) clutters up the dictionary with a lot of class names that will probably never again be referenced, and (3) defines only a single, rigidly constrained kind of "file". Let us explore how one could define a generic "file" in ADAMS.

Consider the following ADAMS declarations

```
REC_LAYOUT is a SET
        << of ATTRIBUTE elements >>

FILE%Z is a SET
        << of RECORD%Z elements,
                where RECORD%Z is a CLASS
                        << RECORD%Z having attributes = %Z >>
        >>
```

In the second ADAMS statement, Z is a macro parameter. It must be instantiated as in:

```
prec belongs to REC_LAYOUT
        << consisting of { p_nbr, name, age, dept, salary } >>

p_file belongs to FILE_prec << >>
```

Now the class name is literally, *FILE_prec*. The instantiation becomes part of the name. The use of macro instantiations provides an economical way of declaring a large number of similar classes. For example, with just to the two basic classes REC_LAYOUT and FILE%Z, one can define an indefinite number of "simple" files.

There is an obvious potential problem with the declaration of *FILE* given above. Its macro parameter could be instantiated with the name of any set, not necessarily a set of attributes. We would like a facility to test for consistency in macro substitution. ADAMS employs a *only if* clause to allow this. The second declaration can be rewritten to read

```
FILE%Z is a SET
          << only if Z.class_of = REC_LAYOUT,
              in which ( ∀ x ∈ FILE%Z) [ x.class_of = RECORD%Z ]
                    where RECORD%Z is a CLASS
                              << having attributes = %Z >>
          >>
```

This example gives a clear illustration of the distinction between the use of *only if, in which,* and *where* clauses. The predicate of a *only if* clause is tested only once—on declaration. If it is not satisfied the declaration is aborted. An *in which* clause becomes part of the persistant declaration. Its predicate is tested repeatedly at runtime whenever objects are manipulated to insure database consistency. Thus when elements are added to a set of the class *REC_LAYOUT* they are tested by the insertion process to make sure that they are of class *ATTRIBUTE.* If the predicate is not satisfied the process is aborted. Finally, a *where* clause denotes an embedded declaration which defines the meaning of a preceding undeclared term.

2.2. Designators

A *name* uniquely designates something. A *name* is an ADAMS symbol that is accessible to the external world (that is an interfacing process) through the dictionary. Most names simply denote specific database objects, such as a file, a relation, or a set. Given the name of the object, interrogation of the dictionary will return all the information a process needs to use the object. Perhaps 90% of all dictionary names are of this category. But there are others. Some names denote co-domains and attributes. Others denote element classes or types. The purpose of many ADAMS statements is to create and/or interpret dictionary entries. Note that sets, classes (or types), co-domains, and attributes are *nameable.* Values in co-domains and some elements are not nameable. All names must be drawn from the pre-defined co-domain *$names.*

A *designator* on the other hand denotes one, or more, things in an ADAMS database. Every name is a designator. A sufficently qualified designator will denote a unique object.

The most common designator is a *variable designator.* In many programming languages these are simply called "variables" or "pointers". For example, in the ADAMS looping construct

```
for all x in qset do ...
```

"x" successively designates individual elements of the named set "qset".

Designators can also be built up using special operators. For example:

```
x.a
```

uses the *attribute application operator* to designate a specific value in a co-domain. In the fragment of code:

```
for all x in qset do
        write (x.a)
```

"qset" and "a" are a named set and attribute respectively. "x" designates some element of qset and "x.a" designates some value in the co-domain of "a".

The *association operator* is denoted by →. It is used to denote a set that has been associated with an ADAMS set or type. For example, to display all of the attributes associated with each element of "qset", we could write

```
for all x in qset do
        for all a in Q→q_attributes do
                write (x.a)
        writeln
```

(Given the way that we had defined the class "Q" in section 2.1.3, we did not really need to use the association operator; "q_attributes" is the name of a uniquely named set.) Consider the second example of 2.1.3, in which we defined a "P_FILE", and suppose that a user process is given only the name of "pfile", a specific object of that class. The following code could be used to display all values contained in "pfile".

```
for all x in pfile do
        for all a in x.class_of→attributes do
                write (x.a)
        writeln
writeln
for all a in pfile→attributes do
        write (a, ’ = ’, pfile.a)
```

Note that in this example the term "attributes" does not uniquely designate any set. The two separate set designators "x.class_of→attributes" and "pfile→attributes" do, however, designate unique sets of attributes. Assuming some reasonable implementation of the *write* operator, we would expect this latter code to generate

```
11111   Smith      36   sales        23,000
22222   Johnson    43   engineering  33,500
33333   Lefler     22   sales        19,000

date_last_mod = 7/14/87
```

For completeness, three additional operators will be mentioned here. All will be developed more fully later. The first two create new symbolic designators from various pieces. The *concatenation operator* is used to explicitly concatenate two substrings to form a single designator as in

<substring1>_<substring2>.

This symbol, underscore ’_’, is ubiquitous in the formation of variable names and identifiers in many traditional languages. The only novelty in ADAMS is that one, or both, of the substrings can be a macro parameter, as in

<class_name>_<macro_parameter>

so that each different macro instantiation creates a different designator.

The *computation operator* converts a computed integer value to symbolic form and concatenates it to form a symbolic name. Given

<symbol>#<integer_1> ,..., <integer_n>#

a standard system function computes a unique integer k from list of n integer values and forms the designator

<symbol>_k

With this operator one can create a rich variety of element designators based on computational processes.

Finally, the *restriction operator* denotes a set designator restricted by another set as in

<set_designator_1>|<set_designator_2>.

It denotes the set of all elements in <set_designator_1> subject to the restriction that they also be elements of <set_designator_2>. In effect | is an intersection operator except that the first operand need not be physically extant as is required by $\cap$. Restriction is primarily used in the performance of inverse operations.

3. Preliminary ADAMS Grammar (BNF)

3.1. Primitive Concepts

```
<id_seg> ::=              <string of letters and/or digits>

<actual_id> ::=           <id_seg> [ _<actual_id> ]

<macro_id> ::=            [ <actual_id> ] % <uc_letter> [ <macro_id> ] [ <actual_id> ]

<class_name> ::=          <actual_id> | <macro_id>

<class_desig> ::=         CLASS | SET | <class_name>

<scope_id> ::=            LOCAL | <persistant_scope>

<persistant_scope> ::=    USER | <user_id> | <group_id> | SYSTEM

<enumerated_set> ::=      { <element_name> [ , <element_name> ] }

<specific_set> ::=        <set_name> | <enumerated_set>

<restricted_set> ::=      <set_designator> ’|’ <set_designator>

<set_designator> ::=      <specific_set> | <instance_name> → <set_synonym> | <restricted_set> | ∅

<function_name> ::=       <attribute_name> | <map_name>

<predicate> ::=           <pred_phrase> [ or <predicate> ]

<pred_phrase> ::=         <pred_factor> [ and <pred_phrase> ]

<pred_factor> ::=         <quantified_p_exp> | <unquantified_p_exp>
```

3.2. Class Definition

```
<class_definition> ::=    <co_domain_def> | <set_def> | <user_class_def>
```

```
<set_def> ::=              <class_name> is a [ <scope_id> ] SET
                                 << [ only if <macro_condition> , ]
                                      of <class_name> elements
                                      [ , <class_definition> ] >>

<user_class_def> ::=   <class_name> is a [ <scope_id> ] <class_desig>
                                 << [ only if <macro_condition> , ][ <class_definition> ] >>

<class_definition> ::=  <def_clause> [ , <def_clause> ... ]

<def_clause> ::=          <pred_restriction> | <association> | <embedded_def>

<pred_restriction> ::= in which <predicate>

<association> ::=         having [ <set_synonym> = ] <specific_set>

<embedded_def> ::=     where <class_definition>
```

3.3. Co_domain Definition

```
<co_domain_def> ::=   <co_domain_name> is a CO_DOMAIN
                                 << of <regular_exp>
                                      [, <co_domain_clause [, <co_domain_clause> ] ] >>

<co_domain_clause> ::=          <udv_clause> | <encode_clause> | <decode_clause>

<udv_clause> ::=          ud_value = <octal_constant>
```

3.4. Instance Specification

```
<instance_specification> ::=    <instance_creation> | <instance_access> | <instance_rescoping> |
                                          <instance_specification>.<map_name>

<instance_creation> ::=          <element_creation> | <set_creation> | <attribute_creation> |
                                          <map_creation>

<element_creation> ::=           <element_name> belongs to <class_name>  << >>

<set_creation> ::=        <set_name> belongs to [ <scope_id> ] <class_name>
                                        << [ consisting of <enumerated_set> ] >>
```

```
<attribute_creation> ::=
                    <attribute_name> belongs to ATTRIBUTE
                          << with image <co_domain_name>
                             [ , <value_assign_clause> ]
                             [ , <udf_default> ]
                          >>

<map_creation> ::=  <map_name> belongs to MAP
                          << with image <class_name>
                             [ , <value_assign_clause> ]
                          >>

<value_assign_clause> ::=
                    <function_name> is assigned   |
                    <function_name> is computed by <value_computation> |
                    <function_name> is external "<external_name>"

<udf_default> ::=         udf = <element_in_co_domain>

<instance_access> ::= use [ <persistent_scope> ] <instance_name> [ of <class_name> ]
                          << <use_restriction> >>

<instance_rescoping> ::=
        rescope <current_scope> <instance_name> of <class_name> to <persistant_scope>
```

3.5. Data Value (co_domain) Access

```
<value_desig> ::=   <element_name> . <attribute_name>

<interpreted_value> ::=   decode ( <value_desig> , <conv_param> ) |
                          <host_language_exp>   |
                          <host_language_variable>   |
                          <host_language_constant>

<value_assignment> ::=
        assign ( <value_desig> , encode ( <interpreted_value> , <conv_param> ))
```

3.6. Object, Set Manipulation

```
<element> ::=  <element_name> | <set_name> | <attribute_name> | <map_name>

<set_op> ::=   ∪ | ∩ | ~
```

<set_exp> ::= <set_designator> <set_op> < set_exp> | <set_designator>

<insertion> ::= **insert** <element_name> **into** <set_degignator>

<deletion> ::= **delete** <element_name> **from** <set_designator>

Note that his grammar may be incomplete.

4. Implementation Examples

ADAMS is intended to be a basic tool with which many database models may be implemented. In this section we illustrate how one may implement

relational,
hierarchical,
network, and
scientific array

databases using ADAMS primitives. One purpose is to demonstrate the general utility of ADAMS concepts. A second purpose is to provide an abundance of ADAMS examples in juxtaposition with known database organizations.

4.1. Implementation of a Relational Model

During the last few years, the majority of formal database theory has been couched in terms of the "relational" model, e.g. [Cod70, MaU83]. It has proven itself to be a flexible context for expressing a wide range of database queries and operations. And because of this flexibility, an increasing number of actual databases are being implemented using relational database software.

The goal of this section is to show that the basic ADAMS constructs are sufficient to support the "relational" approach to database design. At the same time we can illustrate use of many ADAMS syntactical constructs in the context of a fairly specific, concrete application. Note, however, that the "relational model" is a *simple* database model. For example, it requires only the basic concepts of *element, set, attribute* and *co-domain.* The *map* concept, which is used only to associate schema with relation tuples that have that schema, is invisible to the user. It has been this very simplicity that has rendered it amenable to formal analysis, and has led to its wide spread use in practice.

We will demonstrate this capability primarily by developing a running example. Recall the old hackneyed, but comfortably familiar "Suppliers-Supplies-Parts" database that can be found in any undergraduate text. It consists of three relations *supplier, supplies,* and *part* which have the following schema:

supplier (s_nbr, s_name, s_city, s_zip)

supplies (s_nbr, p_nbr)

part (p_nbr, description, unit_price)

We assume that *s_nbr* and *p_nbr* are keys of the *supplier* and *part* relations respectively.

In the following subsections we will illustrate the ways in which the primitive ADAMS concepts can used. The particular syntax of these illustrations is suggestive, but by no means definitive.

4.1.1. User Type Definition

User type definition is a meta-linguistic construct which gives any user the ability to introduce new terms into the ADAMS language. For example:

```
SCHEMA is a SET
          << of ATTRIBUTE elements >>

RELATION%Z is a SET
          << only if  %Z.class_of = SCHEMA
              of %Z_TUPLE elements
                    where %Z_TUPLE is a CLASS
                              << having attributes = %Z >>,
          >>
```

Now *SCHEMA* will denote a restricted kind of set consisting only of elements of type ATTRIBUTE. A *relation* is a set of elements (or tuples) on each of which a specified set of attributes (called the schema of the relation) is defined. Thus to create any "relation", there must first exist a specific named set of attributes to serve as its schema. In the definition of RELATION above, the macro parameter Z will be replaced by the name of the set serving as the schema. That named set is linked with the elements of the relation by a "has" clause.

Since the elements of any set must be typed, we use the *concatenation operator,* _ to generate a (presumably) unique class name *%Z_TUPLE* in which %Z will be replaced with its instantiation.

In the next three ADAMS statements, we first create a specific schema *rel_atts* of three distinct attributes, *a1, a2,* and *a3.* Then we create two different relations *r* and *s* which have identically the same schema.

```
rel_atts belongs to SCHEMA
        << consisting of { a1, a2, a3 } >>

r belongs to RELATION(rel_atts)

s belongs to RELATION(rel_atts)
```

r and *s* denote specific sets of elements, and may be manipulated as such. Since their elements are of the same type, operations such as

$$r \cup s \qquad\qquad r \cap s$$

will be well defined.

Undoubtedly these two important user-defined types *SCHEMA* and *RELATION* would be entered into the global dictionary to become system recognized types. But they need not be. They could be local types that are valid only within the declaring process.

4.1.2. Co-Domain Definition

A co-domain is created by giving a regular expression (using LEX syntax) that describes the set of values and then naming the set. For example:

```
s_nbrs is a CO_DOMAIN
        << of  [A-Z][0-9]{5} >>

string20 is a CO_DOMAIN
        << of  [A-Za-z0-9 -]{0,20} >>

zip is a CO_DOMAIN
        << of  ([0-9]{5} | [0-9]{5}-[0-9]{4}),
           udf = 'e' >>

money is a CO_DOMAIN
```

```
<< of ($ | + | - | ∈)[0-9]{1,10}(.[0-9]{2} | ∈) >>
```

Here we have established the co-domain *s_nbrs* consisting of all 6 symbol strings whose first character is an upper case letter and whose last five characters are digits. Using a COBOL syntax, this co-domain would have been described by PICTURE A99999. *string20* denotes the set of all strings of 20 (or fewer) characters over the alphabet consisting of upper and lower-case letters, digits, blank, and hyphen. The co-domain *zip* permits either the old 5 digit zip codes or the newer 9 digit codes with an embedded hyphen. Finally, the *money* co-domain allows 1 to 10 "dollar" digits, optionally followed by a decimal point and 2 "cents" digits and/or a preceding dollar sign or plus or minus sign. Notice that this definition does not permit embedded commas.

All domains are assumed to have an appended '∈' or udf (undefined) element. In the case of *zip* a specific element, the character 'e', was designated as the udf element. In all the others, a default system udf element was assumed.

4.1.3. Attribute Definition

New attributes are created by simply naming the attribute, designating the co-domain to which it maps, and indicating how attribute values are obtained. This kind of definition is mathematically incomplete because we do not designate the set on which the functional map is defined. In the course of many database operations one creates a new set of elements, on which various attributes will be defined. Thus an *attribute* may be a functional map from many different sets into a single co-domain.

New attributes can be created in the following simple manner.

```
s_nbr belongs to ATTRIBUTE
        << with image s_nbrs, s_nbr is assigned >>

s_name belongs to ATTRIBUTE
        << with image string20, s_name is assigned >>

s_city belongs to ATTRIBUTE
        << with image string20, s_city is assigned >>
```

```
  s_zip belongs to ATTRIBUTE
            << with image zip, s_zip is assigned >>

    p_nbr belongs to ATTRIBUTE
            << with image p_nbrs, p_nbr is assigned >>

    description belongs to ATTRIBUTE
            << with image string20, description is assigned >>

    unit_price belongs to ATTRIBUTE
            << with image money, unit_price is assigned >>
```

In all the preceding examples the attribute values are *assigned* as in traditional database operations. It is possible to declare that attribute values are *computed.*

4.1.4. Creating a Relational Model with ADAMS

Let us assume that generic constructs of *SCHEMA* and *RELATION* have been created by declarative statements as shown in the preceding section 4.1.3. And assume that specific attributes and co-domains exist as a result of statements of the form:

```
    s_nbrs is a CO_DOMAIN
            << of [A-Z][0-9]{5} >>

    s_nbr belongs to ATTRIBUTE
            << with image s_nbrs, s_nbr is assigned >>
```

Co-domains such as *string20, zip,* and *money* are so common that we may assume they have persistant system wide status. (ADAMS is a permanent entity, so its philosophy quite different from more traditional programming languages. Declarations made several years ago in one process may be saved in a persistant dictionary, and used repeatedly by other processes.)

Once the preliminary declarations have been completed, one can actually create the database itself.

```
    s1 belongs to SCHEMA
            << consisting of { s_nbr, s_name, s_city, s_zip } >>

    supplier belongs to RELATION (s1) << >>

    b belongs to SCHEMA
```

```
<< **consisting of** { p_nbr, description, unit_price } >>

parts **belongs to** RELATION (b) << >>

s_s_p **belongs to** SCHEMA
        << **consisting of** { s_nbr, p_nbr } >>

supplies **belongs to** RELATION (s_s_p) << >>
```

In the first statement, the type *SCHEMA* has been previously defined to be a set of attribute(s). Thus *s1* is the name of a specific *schema* instance which consists of precisely the four attributes *s_nbr, s_name, s_city,* and *s_zip.* Attributes could be added to, or deleted, from this schema set at some later time.

The second statement establishes the specific *relation* named *suppliers.* That is, *suppliers* denotes an arbitrary set whose only constraint is that the attributes of *s1* (the RELATION macro parameter) are defined on all of its elements. Since *supplier* has just been created, and no elements have been explicitly added to it, it is *empty.*

It should be apparent that the preceding ADAMS statements have a) re-defined the basic relational terminology in terms of the ADAMS primitives, and b) then created and empty supplier-supplies-parts database. But we have deliberately chosen rather awkward instance names. Our choice *s1, b,* and *s_s_p* as the schema for *supplier, parts,* and *supplies* is not at all nmenonic. Our goal is to force the reader to consciously establish the relationships between them.

The wordiness of these expressions should be an anathema to most database designers and/or programmers. Indeed, we would expect to see many more concise forms of expression such as,

```
schema s1 is { s_nbr, s_name, s_city, s_zip }.

relation suppliers has schema s1.
```

where these more concise forms could be implemented in the language itself, as macro substitutions over the language, or as procedures. This is the reason for regarding all of the examples in this paper only as illustrative (not definitive) of ADAMS syntax.

4.1.5. Views

In the preceding sections we used the following class definition to create the concept of a "relation":

```
RELATION%Z is a SET
        << only if  Z.class_of = SCHEMA
           of %Z_TUPLE elements
                where %Z_TUPLE is a CLASS
                        << having attributes = %Z >>,
        >>
```

This economical definition associated (the instantiation of) Z, which must be a SCHEMA or set of attributes, with each element x in a relation of that scheme.

Every user has precisely the same "view" of this relation, since only one schema is associated with its elements. But there are times when one wants different users to have different views of the same data. For example, the owner of our running "supplier_supplies_parts" database may want all his salesmen to know the unit price of parts in the inventory, but not necessarily their cost. (Perhaps he is embarassed by the extreme markup!) A relation with two separate views could be created as follows

```
PART is a CLASS
        << having attributes = { p_nbr, description, unit_price },
           having owners_view = { p_nbr, description, unit_price, unit_cost }
        >>
PART_REL is a SET
        << of PART elements >>
```

The customary ADAMS expression

```
for all x in parts do
        for all a in x->attributes do
                write (decode(x.a))
        writeln
```

will only display the *p_nbr*, *description* and *unit_price* of every element in the relation. Only those users in possession of the *owners_view* can also display *unit_cost.*

The ability to associate several sets with any ADAMS element provides a flexible way to implement multiple views in a database.

4.2. Using Maps

Before illustrating the implementation of either the hierarchical or network models we must develop the ADAMS concept of a *map* more fully. The relational model does not require a "map" concept, which is at once both its great strength and its weakness. Mapping concepts are unfamiliar to many database users; so a database model which is based on only flat tables is much easier to explain and to visualize [Cod70]. But invariably relationships must be created between data sets; and the relational join is not always the most effective way of implementing the relationship.

An *attribute* function defined on an ADAMS element functionally associates a single co-domain value with that element. A *map* defined on an ADAMS element functionally associates a single ADAMS element with that element. That is, attributes are functions from a class of ADAMS elements into co-domains; maps are functions from a class of ADAMS elements into another class of ADAMS elements. Commonly the image element of a map is itself a set element; thus while maps are properly single valued, they can nevertheless be used to represent one to many mappings.

A map is declared by naming it, designating its image class, and indicating how image elements are determined, as in

```
f belongs to MAP
        << with image Q, f is assigned >>
```

Here, the image under the map *f* must be a single element in the class Q, and for any element x, f(x) must have been previously set with an assignment statement. Alternatively, one might declare

```
g belongs to MAP
          << with image Q_SET, g is external "g_proc" >>
```

Any image under the map *g* will be a (possibly empty) set of Q elements, that is a single Q_SET. The set of image elements will be procedurally determined by the externally coded procedure "g_proc".

Maps are associated with elements in a class, just as attributes are associated with the elements of a class. In the following example, any element of class P has the two map functions *f* and *g* defined on it, as well as the three attributes *a1, a2,* and *a3.*

```
P is a CLASS
          << having { f, g } ,
               having { a1, a2, a3 }
          >>
```

Assume the ADAMS statement

```
x belongs to P << >>
```

Now *x.f* will denote an element from the class Q. Note that we use the same application operator, to apply maps that we use to apply attribute functions to an element. The expression *x.f.a* denotes the attribute *a* defined on the Q type element denoted by *x.f.* Or, since *g* maps to a Q_SET of Q type elements, we could used the fragment of code

```
for all y ∈ x.g
          write (y.a)
```

to display the *a* attribute values of every element y that is the set which is the image of x under the *g* map.

As before, we may choose to provide a synonym for either the set of associated maps or attributes as below.

```
P is a CLASS
        << having maps = { f, g } ,
            having attributes = { a1, a2, a3 }
        >>
```

In this example, we have used "maps" to be a synonym for the set of associated maps, and "attributes" to be a synonym for the set of associated attributes. This is simply a convention which allows us to access the set of of *all* associated maps (or attributes) using the association operator

$$x \rightarrow maps \qquad\qquad x \rightarrow attributes$$

without having to remember them specifically.

4.2.1. Inverses

Let y be the image of an element x under the map f; that is, y = x.f. The class of y is well defined since by declaration the image space of f must be a single class.

The *inverse image* of y under f is the set X of all elements $\{x_1, x_2, ..., x_n\}$ such that $x_k.\mathrm{f} = y$. That is,

$$X = y.\mathrm{f}^{-1} = y.\mathrm{f}(inv) = \{\, x_k \mid x_k.\mathrm{f} = y \,\}$$

Of course, x ∈ X. But there may be many other elements as well. The problem is that X is not a well defined set, in the ADAMS sense. Since the map f can be associated with (defined over) several different classes of elements, X need not consist of only elements of the same class.

To be meaningful any inverse image set y.f(**inv**) must be restricted to elements of the same class. For computational efficiency reasons ADAMS is even more stringent. *Inverse functions must be restricted to a single extant set to be valid,* as in

x := y.f(**inv**) | z

where *x* and *z* must be instance sets of the same class, and *f* must be defined on elements of these sets. This constraint might seem to be overly strict; but recall that implementing the inverse function operator is precisely the same as solving the general database retrieval problem. And all database languages retrieve elements from a specified file, relation, or set.

For example, suppose we declared a class STUDENT satisfying the relational schema

```
STUDENT:      (SID, Name, Major, Grade_level, Age)
```

Then the following query expressed in SQL syntax

```
SELECT        SID, NAME, MAJOR, GRADE_LEVEL, AGE
FROM          STUDENT
WHEREMAJOR = ’MATH’
```

would be expressed in ADAMS as

```
x belongs to STUDENT << >>

x := ’math’.major(inv) | student
for all y in x do
        write (y.sid, y.name, y.major, y.grade_level, y.age)
```

4.3. Implementing a Hierarchical Model

One common usage of maps is the implementation of "set valued" attributes, or repeating groups, as occur in a hierarchical database. The "personnel file" example of section 2.1.3 can be extended to include a salary history with each record in the file as shown below.

```
DATA_FIELDS is a SET
        << of ATTRIBUTE elements >>

master_rec_data belongs to P_FIELDS
        << consisting of { p_nbr, name, age, dept } >>

salary_rec_data belongs to P_FIELDS
        << consisting of { b_date, e_date, salary } >>

SALARY_REC is a CLASS
        << having attributes = salary_rec_data >>

SALARY_HISTORY is a SET
        << of SALARY_REC elements >>

salary_history belongs to MAP
```

```
<< with image SALARY_HISTORY, salary_history is assigned >>

P_REC is a CLASS
        << having attributes = master_rec_struct,
           having maps = { salary_history }
        >>

P_FILE is a SET
        << of P_REC elements,
           having attributes = { date_last_mod }
        >>
```

This definition is equivalent to the COBOL data definition

```
01 data record is
        02      p_nbr               PICTURE.
        02      name                PICTURE.
        02      age                 PICTURE.
        02      dept                PICTURE.
        02      salary_history      OCCURS n TIMES.
                03      b_date      PICTURE.
                03      e_date      PICTURE.
                03      salary      PICTURE.
```

except that we have omitted the co-domain definitions of the individual attributes, *p_nbr*, ..., *salary* which in COBOL would have been handled by the indicated PICTURE clauses. Also our definition provides for an attribute *date_last_mod* which is associated with the "set" (or file) as a whole, and not with any particular element (or record) in it.

4.4. Implementation of a Network Model

The network model, as defined by the CODASYL/DBTG report [AAA75, NNN73], explicitly used the concept of a "set", but only as the set of elements which were the image of a one to many map. But the term "map" was never used. Instead, every set was "owned" by an element of some other class. That owner element was the "pre-image" of the map. The CODASYL (or network) model implements a one-to-many map (for each domain element there is a set (of many) elements that its owns). ADAMS implements many-to-one maps. Consequently, ADAMS descriptions must often include the extra class declaration, of a *set* of image elements. After this is done simple network models are easy to describe in an ADAMS syntax, using a construction very similar to the hierarchical decomposition of the preceding section.

The following ADAMS statements will declare a CODASYL/DBTG schema

A:(a1, a2, a3)

B:(b1, b2)

C:(c1, c2, c3, c4)

where the characteristics of a1,...,c4 are defined elsewhere. In this simple network data base, records of type A "own" sets of type B and of type C. Each record of type B also "owns" records of type C.

```
DATA_FIELDS is a SET
          << of ATTRIBUTE elements >>

a_fields belongs to DATA_FIELDS
          << consisting of { a1, a2, a3 } >>

b_fields belongs to DATA_FIELDS
          << consisting of { b1, b2 } >>

c_fields belongs to DATA_FIELDS
          << consisting of { c1, c2, c3, c4 } >>

C_REC is a CLASS
          << having fields = c_fields >>

C is a SET
          << of C_REC elements >>

c_set belongs to MAP
          << with image C, c_set is assigned >>

B_REC is a CLASS
          << having fields = b_fields,
              having image_sets = { c_set }
          >>

B is a SET
          << of B_REC elements >>

b_set belongs to MAP
          << with image B, b_set is assigned >>
```

```
A_REC is a CLASS
            << having fields = a_fields,
               having image_sets = { b_set, c_set }
            >>
    A is a SET
            << of A_REC elements >>
```

At this point the definition of the database structures exist, but no sets of type A, B, or C actually exist.

In the CODASYL/DBTG report, a set can have several "owners" provided each owner is of a different type. The same is true in ADAMS, except that we relax the latter restriction. ADAMS functions can be defined on distinct sets of the same type. For example, the map *c_set* is defined on two different sets of elements, which in this case are also of different types A and B. Consequently, it is hard to define just what is meant by the "inverse image" of an element (or subset) in a set of type C under the "c_set" map. For this reason, ADAMS requires every inverse image operation to be restricted to a single set, as described in 4.2.1. We avoid this potential confusion in the following alternative declarations, which make use of the internal definition *where* construct. There are two distinct "c_set" maps.

```
C is a SET
        << of C_REC elements
                where C_REC is a CLASS
                        << having fields = { c1, c2, c3, c4 } >>
        >>

B is a SET
        << of B_REC elements
                where B_REC is a CLASS
                        << having fields = { b1, b2 },
                           having maps = { c_set1 }
                               where c_set1 belongs to MAP
                                       << with image C, c_set1 is assigned >>
                        >>
        >>

A is a SET
        << of A_REC elements
                where A_REC is a CLASS
                        << having fields = { a1, a2, a3 },
                    having maps = { b_set, c_set2 }
                              where b_set belongs to MAP
                                    << with image B, b_set is assigned >>
                              where c_set2 belongs to MAP
                                    << with image C, c_set1 is assigned >>
                        >>
            >>
```

Is this "top-down" declaration more understandable?

In the network approach to database design, many-to-many mappings were represented by introducing an extra class of *intersection* records. Basically, a many to many relationship such as "enrolled" between STUDENT and COURSE in the complex network

FACULTY: (name, rank, dept)

advisor

instructor

STUDENT: (name, major, student_nbr)

enrolled

COURSE: (c_nbr, c_name, term)

must be reduced to two separate one to many mappings, "enrolled_in" and "enrollment" as in

FACULTY: (name, rank, dept)

advises

teaches

STUDENTS: (name, major, student_nbr)

COURSES: (c_nbr, c_name, term)

student

course

ENROLLMENT: (grade)

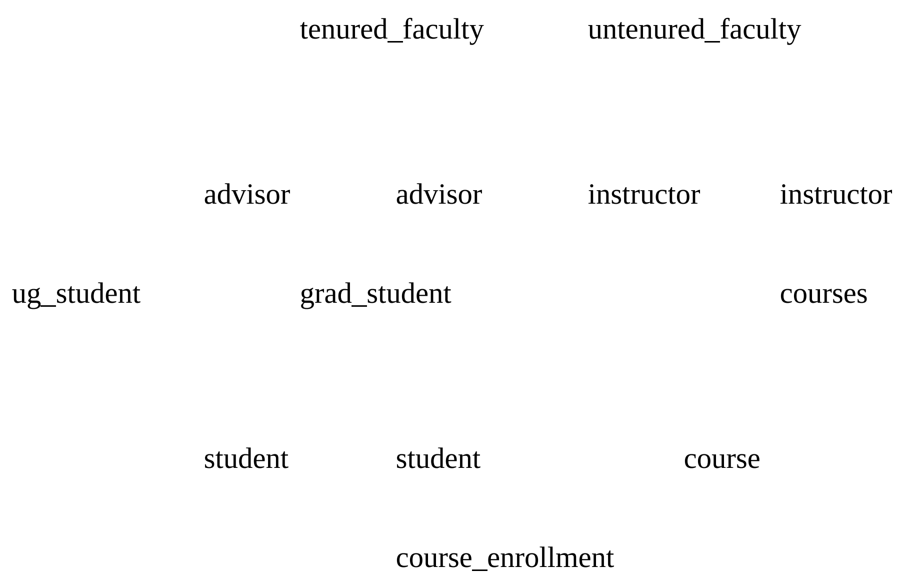
where in a network implementation ENROLLMENT records would be intersection records. Since ADAMS implements its maps as many to one functions, these ENROLLMENT elements are truely *ordered pairs* in a binary relation. Each element of type ENROLLMENT has a *student* "pointer", a *course* "pointer", and a *grade* attribute. A network structure of this nature would be declared by the following ADAMS statements.

```
FACULTY_REC is a CLASS
        << having data_fields = { name, rank, dept },
           having maps = { teaches, advises }
        >>

FACULTY is a SET
        << of FACULTY_REC elements >>

instructor belongs to MAP
        << with image FACULTY_REC, instructor is assigned >>

advisor belongs to MAP
        << with image FACULTY_REC, advisor is assigned >>

COURSE_REC is a CLASS
        << having data_fields = { c_nbr, c_name, term },
           having maps = { instructor }
        >>

COURSES is a SET
        << of COURSE_REC elements >>


STUDENT_REC is a CLASS
        << having data_fields = { name, major, student_nbr },
           having maps = { advisor }
        >>

STUDENTS is a SET
        << of STUDENT_REC elements >>

student belongs to MAP
        << with image STUDENT_REC, student is assigned >>

course belongs to MAP
        << with image COURSE_REC, course is assigned >>

ENROLL_REC is a CLASS
        << having data_fields = { grade },
           having maps = { student, course } >>

ENROLLMENT is a SET
        << of ENROLL_REC elements >>
```

No specific sets of type FACULTY, STUDENT, COURSE, or ENROLLMENT have been created. There could be several in each class. For example, the ADAMS statements

> tenured_faculty **belongs to** FACULTY << >>
>
> untenured_faculty **belongs to** FACULTY << >>
>
> courses **belongs to** COURSES << >>
>
> ug_student **belongs to** STUDENTS << >>
>
> grad_student **belongs to** STUDENTS << >>
>
> enrollment **belongs to** ENROLLMENT << >>

would create a structure of the form



All of these sets are empty. Elements can be inserted into the sets with a sequence such as

> x **belongs to** STUDENT_REC << >>
>
> **assign** (x.name, **encode**('Smith')).
>
> **assign** (x.major, **encode**('computer science')).
>
> **assign** (x.student_nbr, **encode**(101234567)).
>
> **insert** x **into** grad_student.

```
y belongs to FACULTY_REC << >>
    assign (y.name, encode('Pfaltz')).
    insert y into tenured_faculty.
    assign (x.advisor, y).
```

This example can be used to provide an excellent illustration of inverse function concepts.

4.5. Partitions

There are a number of instances in which it is desirable to partition the elements of a database. For example, in a distributed system one often partitions a collection of database items, putting some of the items on one processor, other items on other processors. Using the rows (or columns) of a matrix to partition the array and provide the basis of a storage organization is another familar example. In this section, we will examine the ADAMS mechanisms for defining and creating *partitions* of database elements. We will also explore the potential for partitioning co-domains.

4.5.1. Partitions of ADAMS elements

Recall that a partition of a set s of elements is a collection of subsets $s_1, s_2, \cdots s_n$ such that

(a) $\cup_{i=1}^{n} s_i = s$

(b) $s_i \cap s_j = \emptyset$, $\forall$ i $\neq$ j.

Thus a *partition* must be a set of sets, and the sets must be pairwise disjoint. It is a matter of taste whether one also wants to require that each subset s_i be non-empty. We explicitly allow empty subsets in a partition.

We can define a partition within ADAMS by the sequence

```
PARTITION%s is a SET
        << of SET elements,
            in which (∀ x  PARTITION%s) [ x ⊆ s
                 and (∀ y  PARTITION%s) [ x ≠ y implies x ∩ y = ∅ ] ]
        >>
```

Since it only satisfies the weakened condition

(a') $\cup_{i=1}^{n} s_i \subseteq s$.

this declaration defines a disjoint union, not a true partition. But this is an operational necessity. Without it, one could not dynamically modify a partitioned set. Inserting an element into the base set s would immediately violate a true "partition" concept, since it could not simultaneously be inserted into one of the member sets of the partition.

Let s be any set of elements and let P_1 be a partition (or disjoint union of subsets) of s. Say $P_1 = \{ s_{11}, s_{12}, \cdots , s_{1m} \}$. Now let P_2 be a second partition of s, say $P_2 = \{ s_{21}, s_{22}, \cdots , s_{2n} \}$. By the *refinement* of P_1 with respect to P_2, denoted **ref(P_1, P_2),** we mean the collection of subsets of s

$$\begin{array}{l} \{ s_{11} \cap s_{21}, s_{11} \cap s_{22}, \cdots , s_{11} \cap s_{2n} , \\ \ \ s_{12} \cap s_{21}, s_{12} \cap s_{22}, \cdots , s_{12} \cap s_{2n} , \\ \qquad \vdots \qquad \vdots \qquad\qquad \vdots \\ \ \ s_{1m} \cap s_{21}, s_{1m} \cap s_{22}, \cdots , s_{1m} \cap s_{2n} \ \} \end{array}$$

Many of these subsets may be empty. Many may have several elements. If it can be shown that each subset $s_{1j} \cap s_{2k}$ of the refinement has at most one element, then s can be regarded as a 2-dimensional array which is indexed by the sets P_1 and P_2. More generally,

> **A set s of elements can be treated as an n-dimensional array if there exists an n-fold refinement ref($P_1, P_2, \cdots , P_n$) such that each subset has cardinality at most one.**

4.5.2. Use of Partitions to Distribute Data

One important use of the partition concept in ADAMS is to explicitly control at the syntactic level the distribution of data representations. Recall from section 2.1 the set of declarations creating *q*, a set of Q-type elements.

```
Q_A is a SET
        << of ATTRIBUTE elements >>

q_attributes belongs to Q_A
        << consisting of { a1, a2, a3 } >>

Q is a CLASS
        << having q_attributes >>

Q_SET is a SET
        << of Q elements >>

q belongs to Q_SET << >>
```

In a *horizontal* distribution of the set *q* some of its elements would be represented at one node (either processor or storage device), while other element would be distributed to other nodes. In many systems (e.g. [ShI85]) horizontal distribution is a *defacto* partition of the set of distributed data elements. The following sequence of ADAMS statements makes partitioning explicit and can be used to control the distribution process.

```
q_partition belongs to PARTITION(q)
        << consisting of { q1, q2, q3 } >>

for all x in q
        if x.a1 ≤ 0
          then insert x into q1
          else if 0 < x.a1 and x.a1 ≤ 100
                  then insert x into q2
                  else insert x into q3.

represent q1.q_attributes at node_a.
represent q2.q_attributes at node_b.
represent q3.q_attributes at node_c.
```

The horizontal partition is based on the value of attribute *a1*. Depending on this value the element *x* of *q* (actually all of its attribute function values) is represented at *node_a*, *node_b* or *node_c*.

In a horizontal distribution of a relation some of its tuples reside at one node, some at another. In a *vertical* distribution, part of each tuple resides at one node, other parts reside at other nodes. Vertical distribution is quite easy in ADAMS; one simply partitions the set of attributes as below. Let the set of Q elements, *q*, be declared as at the beginning of this sub-section.

```
a_partition belongs to PARTITION(q_attributes)
          << consisting of { a_part, b_part } >>

insert a1 into a_part.
insert a2 into a_part.
insert a3 into b_part.

represent q.a_part at node_a.
represent q.b_part at node_b.
```

Here the *a1* and *a2* values associated with every element of the set *q* are represented at *node_a,* while *a3* values are represented at *node_b.*

We will explore other applications of the partition concept in the next section concerned with array data.

4.5.3. Co-domain Partitions

Co-domains can not be partitioned in the same way that a set of elements can be partitioned and dynamically re-partitioned. But a static co-domain partition can be approximated through careful definition. The n subsets of the partition are declared first. The co-domain is then defined as the finite union of these n subsets. Since each of the subsets is a regular set, their union is also regular. For example,

```
name_1 belongs to CO_DOMAIN
          << consisting of [A-D]{any_letter} >>

name_2 belongs to CO_DOMAIN
          << consisting of [E-L]{any_letter} >>

name_3 belongs to CO_DOMAIN
          << consisting of [M-P]{any_letter} >>

name_4 belongs to CO_DOMAIN
          << consisting of [Q-Z]{any _letter} >>

last_name belongs to CO_DOMAIN
```

<< **consisting of** {name_1} | {name_2} | {name_3} | {name_4} >>

is a definition of the co-domain *last_name* as the union of 4 roughly equal sized sub-domains.

4.6. Representation of Array Data

A co-domain consists of simple "scalar" values. (Actually, a co-domain consists only of strings as was explained in section 1.6.) There is no provision for storing or retrieving an array of values. But arrays are an important class of data. In ADAMS there are three very different ways of accomodating array data.

4.6.1. Pseudo Arrays

A *pseudo array* is simply an ADAMS set on which an array structure is imposed by the processing procedure. For example, let *a_set* denote any set of elements, all of which have the attribute *a* defined on them. Then the fragment of Pascal-like code

```
i := 1
for all x in a_set do
        begin
        A[i] := decode(x.a)
        i := i+1
        end
```

will create an array *A* using the set of values denoted by *a_set.* The use of pseudo arrays is a surprisingly simple and straight forward way of using ADAMS to store and retrieve array structured data. The data is stored and retrieved from external storage as a simple set. A loop, such as that above, maps it into internal storage for processing. After processing an inverse loop maps the array back into its set format for subsequent storage.

In the example above the array was represented as a set of elements with a single attribute, *a,* defined on each. Because of the symmetry in ADAMS which allows us to process sets of elements, we can construct a dual formulation in which the array is represented by a single element with a set of attributes. In this dual example, let

A_ELEMENTS **is a** SET

```
a_elements belongs to SET
        << consisting of { a1, a2, a3, a4, ... , an } >>
```

in which all the attribute functions have the same co-domain. Now, in an almost identical loop to that above

```
i := 1
for all a in a_elements do
        begin
        A[i] := decode(x.a)
        i := i+1
        end
```

the local process array ***A*** **is created by looping over a set of attributes. We prefer to regard an array as a single element with multiple attributes defined on it. The remaining examples will make this preference evident. Either way of representation seems satisfactory; our approach explicitly lifts the common array restriction that all elements be homogeneous (of the same type). Relaxation of this restriction may, or may not, be advantageous.**

While these pseudo arrays are a simple, and often useful, way of representing arrays, there is a significant drawback. The approach relies on an implementation detail. In an ADAMS implementation, *the order of elements in a for each loop over a fixed set is invariant.* The order in which elements of a set are accessed can not be predicted a priori, but once the order is established it does not change. Reliance on such implementation conventions is always dangerous.

4.6.2. Computed Identifiers

The customary notation **a[i,j]** used to denote subscripted variables in traditional programming languages such as Fortran and Pascal is a marvel. Stored scalar values are accessed by arbitrary symbolic identifiers called "variables". However, the variable **a[i,j]** is not a symbolic identifier, it is a "computed" identifier. An evaluation function is applied to the integer variables Even if the variable expression is **a[1,3]** it is still "computed", using two constants, 1 and 3, instead of variables, i and j. It is not a symbolic identifier in the way that a_{13} or a_{ij} are symbolic mathematical identifiers.

The ability to "compute" identifiers is a well known boon to anyone who has ever written code to perform a matrix multiplication using a *for* loop. But it comes at a price. The computational ability requires a standardized storage convention. All subscripted variables must be represented the same way, and their dimensions must be declared in advance. (Element addressing schemes which support variable dimensioning normally use linked lists and involve both a computational procedure and a search procedure.) Instead of using array notation to compute a storage location in an address space, ADAMS employs the notation

<symbol>#<integer>,...,<integer>#

to "compute" an identifier in a name space. The resultant identifier can be arbitrary; but the convention used by ADAMS is to compute an integer *k* based on the sequence of integer parameters and simply concatenate the symbolic equivalent of k to <symbol> forming *<symbol>_k.* Thus for example, *q_set#17#* would become simply *q_set_17.* Determining the appropriate integer value for *k* in the case of two, or more, integer parameters is more difficult. Of course, one wants a unique integer *k* for every possible combination of integer parameters.

One of the more popular computations for doubly subscripted identifiers is the familiar "column major" formula used by many compilers. In this one lets

$$k = (i - 1) + ub_i * (j - 1)$$

denote an offset (from the beginning of the array) locating the element $a_{i,j}$ in the array. This formula presumes that $0 \leq i \leq ub_i$. The range of possible i values must be specified in advance. A better alternative formula which is independent of ub_i is based on the diagonal enumeration

$$k = [\ (i+j-1)(i+j-2)/2\ ] + i$$

in which every lattice point (i,j) of the positive quadrant is uniquely numbered by the integer k.

Now, given an expression of the form

```
<symbol>#i,j#
```

a symbolic version of the integer k which is an integer function of i and j can be concatenated with <symbol> to form a unique identifier.

An example may help clarify this way of representing arrays. We will describe a class of doubly subscripted real arrays in which each element value is accessed by a different attribute function. We begin with two declarations

```
real_elements belongs to SET
        << in which (∀ x ∈ real_elements) [ x.class_of = ATTRIBUTE
                        and x.co_domain_of = REAL] >>

REAL_ARRAY is a CLASS
        << having real_elements >>
```

The set *real_elements* is an empty set. Thus any instance of a REAL_ARRAY will have no elements. We correct this by dynamically declaring what elements (actually attributes) will be recognized in a REAL_ARRAY.

```
for i := 1 to max_row
        for j := 1 to max_col
                begin
                v#i,j# belongs to ATTRIBUTE
                        << with image REAL, v#i,j# is assigned >>
                insert v#i,j# into real_elements
                end
```

Note that in this code fragment, each occurance of the expression *v#i,j#* will be replaced by whatever identifier it computes. Now the class REAL_ARRAY denotes objects which have *max_row * max_col* associated attribute functions. An ADAMS statement

```
a belongs to REAL_ARRAY << >>
```

would create an instance of REAL_ARRAY called *a*. A sequence of executable code such as

```
for i := 1 to n_rows
        for j := 1 to n_cols
                begin
                temp := decode(a.v#i,j#) + decode(b.v#i,j#)
                assign (c.v#i,j#, encode(temp))
                end
```

would add arrays *a* and *b* to form *c* elementwise. However we would strongly discourage such code. It would be horribly slow! ADAMS is a storage and retrieval language, not a computational language. ADAMS should only be used to access arrays; they should be copied onto host language structures for subsequent processing.

4.6.3. Access by Index Partitions

An alternative way of creating array structured data sets is by forming partitions. As developed in section 4.5, if one creates n mutually orthogonal partitions of a set such that n-fold intersection of subsets from each of the partitions contains at most one element, then the set can be treated as an n-dimensional array. Each partition can be regarded as an index of the array. And the name of a subset within the partition is an index value.

An example may be the clearest way of explaining this. Recall the preliminary declarations from the preceding section.

```
real_elements belongs to SET
        << in which (∀ x ∈ real_elements) [ x.class_of = ATTRIBUTE
                        and x.co_domain_of = REAL] >>

REAL_ARRAY is a CLASS
        << having real_elements >>
```

As before we will impose the array structure on the set of attributes so that each array is a single ADAMS element. (We could as well create a set of elements with a single attribute function and impose the array structure on the elements.) We begin by creating two separate partitions called *rows* and *columns* respectively.

```
rows belongs to PARTITION(real_elements) << >>

    columns belongs to PARTITION(real_elements) << >>

    for i:= 1 to m
            row#i# belongs to real_elements << >>
            insert row#i# into rows.

    for j := 1 to n
            col#j# belongs to real_elements << >>
            insert col#j# into columns.

    for i := 1 to m
            for j := 1 to n
                    k := [(i + j - 1)(i + j - 2)/2] + i
                    a#k# belongs to ATTRIBUTE
                            << with image REAL,
                               a#k# is assigned >>
                    insert a#k# into real_elements.
                    insert a#k# into row#i#.
                    insert a#k# into col#j#.
```

At this point the set of m x n attribute functions *a_1, a_2, ...* have been defined and subdivided into the two partitions, *rows* and *columns.* Now *row#5#* denotes the "5-th row" of any array element on which the attribute functions of "real_elements" are defined. And *(row#i#* $\cap$ *col#j#)* denotes the [i,j]-th attribute function in "real_elements". By construction the intersection of any set of the *rows* partition with any set of the *columns* partition must yield at most a singleton element. So this refinement is equivalent to a doubly subscripted array.

All this seems unnecessarily complex. However, one would expect the actual complexity to be hidden from the user who would use a much simpler notation convention. And one does gain something from such an array access notation.

First, while we have used the computed symbol operator # to numerically denote the subsets in "rows" (and "columns"); it was not necessary. Any subset names would have been sufficient. In this manner a multi-dimensional array can be indexed by any discrete set, not just integers. The symbolic name could be generated by an a.i. procedure, thereby dynamically imposing a sparse array structure on an event space.

Second, any array so defined can be easily distributed to various nodes, either by "rows" or by "columns".

Finally, arrays so defined are dynamic. The current array bounds of *m* and *n* were used to create an initial set of attribute function names and partition subset names. But we can create more. We can dynamically enlarge (or shrink) the real array so defined. If *x* and *y* defined by

```
x belongs to REAL_ARRAY << >>

y belongs to REAL_ARRAY << >>
```

are real m x n arrays, they can be expanded to m’ x n arrays (m’ > m) by the statements

```
for i := m to m’
          row#i# belongs to real_elements << >>.
          insert row#i# into rows.

for i := m to m’
          for j := 1 to n
                    k := [(i + j - 1)(i + j - 2)/2] + i
                    a#k# belongs to ATTRIBUTE
                              << with image REAL,
                                  a#k# is assigned >>
                    insert a#k# into real_elements.
                    insert a#k# into row#i#.
                    insert a#k# into col#j#.
```

These newly declared array attribute functions are now defined for all existing ADAMS elements (such as *x* and *y)* which have *real_elements* as an associated set of attributes. Of course, the attribute function values will be NULL until they are explicitly assigned.

The purpose of ADAMS is simply to store and access data in various forms. As such we expect its declarations to be quite static. Many declarations will be valid for years; long after the declaring process has ceased to be of any use. Static declarations are the norm. But it is reassuring to know that declarations can also be dynamically modified.

5. References

[AAA75] *CODASYL Data Base Task Group Report, 1971*, ACM, New York, 1975.

[ACO85] A. Albano, L. Cardelli and R. Orsini, Galileo: A Strongly Typed Interactive Conceptual Lanugage, *ACM Trans. Database Systems 10*,2 (June 1985), 230-260.

[BBK87] F. Bancilhon, T. Briggs, S. Khoshafian and P. Valduriez, FAD, a Powerful and Simple Database Language, *Proc. 13th VLDB Conf.*, Brighton, England, Sep. 1987, 97-105.

[BuA86] P. Buneman and M. Atkinson, Inheritance and Persistence in Database Programming Languages, *Proc. ACM SIGMOD Conf. 15*,2 (May 1986), 4-15.

[Cod70] E. F. Codd, A Relational Model for Large Shared Data Banks, *Comm. ACM 13*,6 (June 1970), 377-387.

[CAD87] R. L. Cooper, M. P. Atkinson, A. Dearie and D. Abderrahmane, Constructing Database Systems in a Persistent Environment, *Proc. 13th VLDB Conf.*, Brighton, England, Sep. 1987, 117-125.

[CoM84] G. Copeland and D. Maier, Making Smalltalk a Database System, *Proc. SIGMOD Conf.*, Boston, June 1984, 316-325.

[MaU83] D. Maier and J. D. Ullman, Maximal Objects and the Semantics of Universal Relation Databases, *ACM Trans. Database Systems 8*,1 (Mar. 1983), 1-14.

[NNN73] *CODASYL Data Description Language*, National Bureau of Standards Handbook 113, U.S. Dept. of Commerce, Washington, DC, 1973.

[SSE87] A. Sernadas, C. Sernadas and H. Ehrich, Object-Oriented Specification of Databases: An Algebraic Approach, *Proc. 13th VLDB Conf.*, Brighton, England, Sep. 1987, 107-116.

[ShI85] D. G. Shin and K. B. Irani, Partitioning a Relational Database Horizontally Using a Knowledge-based Approach, *Proc. ACM SIGMOD Conf.*, Austin TX, Dec. 1985, 95-105.

[Shi81] D. W. Shipman, The Functional Data Model and the Data Language DAPLEX, *ACM Trans. Database Systems 6*,1 (Mar. 1981), 140-173.

Table of Contents