Scoping Persistent Name Spaces in ADAMS

J.L. Pfaltz,
J.C. French,
J.L. Whitlatch

TR-88-03
June 28, 1988

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

Scoping Persistent Name Spaces in ADAMS

John L. Pfaltz
James C. French
Jenona L. Whitlatch

IPC-TR-88-003
June 28, 1988

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract:

ADAMS is based on five primitive concepts: Attribute, co-domain, element, map, and set. Each instance of these primitives is a named entity to which user references resolve. Thus, the concept of naming and name resolution is crucial to the effective management of data in ADAMS.

This paper describes the hierarchical name space proposed for the adams database environment. The reasons for choosing a hierarchical model are examined and the difficulties engendered by this choice are explored.

This research was supported in part by JPL Contract #957721

1. Introduction

The concept of *naming* is essential to the process of communicating ideas and information. But what is a "name"? We should be able to agree that a name is a sequence of symbols drawn from some alphabet, a word, or more simply a string. But with this we may have reached the limit of communal consensus. The real issue is "what is the function of *names* in communication?". This must be resolved before we can develop a meaningful treatment of names in computer languages.

Traditionally, names are used to identify various kinds of things in our universe of discourse. They are *nouns.* When the identification is of a unique entity, the name is said to be a *proper noun* and it is commonly capitalized (in English) to call attention to this role. If it is not a single unique entity that the word identifies—it might denote a group of entities, a concept, or a mass object—we call it a *common noun.* It seems that the unique identification of a single thing, or object, is what most people mean when they speak of *name, naming* or *nameability.* Unique identification is also the customary role of *names* in programming languages. We use variable names and named constants. Procedures and tasks can be named. But when the word denotes a class of things, for instance a data type, we seldom refer to it as a name.

But there is a danger in tying the the concept of a name too closely with that of the unique identification of a single entity or object. First, Khoshafian and Copeland point out in their excellent paper [KhC86], there are many ways of identifying objects of interest besides naming them. One should avoid treating *naming* and *identification* as if they were synonyms. Second, the concepts of *uniqueness* and *singularity* are difficult to establish rigorously. They are very context dependent. Formally, one has to deal with hierarchies of equivalence relations. Less formally, one must come to grips with issues such as "when is a collection of several things, such as tuples or records, a single nameable thing, such as a relation or file?"; or "must any aggregate, such as all files unused since 1 January 1987, that I choose to *name* be regarded as a single entity and be

implementable as such?". These are not unsurmountable issues; but clearly they can be messy. Third, as we develop computer languages that are capable of handling persistent objects, e.g. [BuA86], we must be able to create and name persistent types, which are not traditional entities and need not be singular.

In ADAMS, the Advanced DAta Management System being developed at the University of Virginia [PSF87], we resolve a number of these issues by regarding as "names" all noun-like words which *denote* in some manner things of interest in our particular universe of discourse which is the implementation of, and access in, databases. Because this will be the thrust in subsequent sections, we toyed with using the title *"Noun Denotation in ADAMS"* for this paper. We did not only because we felt that the concepts of "names" and "scoping" would carry more significance to potential readers. Throughout this paper we will use *name* as if it were a synonym for a generic *noun* and not necessarily a "proper" noun.

Generalizing the nature of the problem to the denotation of noun-like words within a computer language helps to avoid needless involvement with the issue of unique entity identification. But many other issues remain. We will sketch the nature of some of these issues in this introduction and then develop the approach taken in ADAMS more fully in subsequent sections.

The *meaning* of a name is that object, or element, or concept, which it denotes. Even though ADAMS may be regarded as an object-oriented database language [CoM84, MSe86], we try to avoid using the term *object.* The things that an ADAMS name denotes are not true objects in the sense of [GoR83, 81]; that is, their manipulation is not restricted to only those methods (or processes) declared for the class. We prefer to use a more neutral term, *element.* But, in the context of this paper, the reader may choose to treat element and object as synonyms.

An element is an instance of a class. Both elements and classes are nameable. In a large community of users it is desirable to allow many names to be reused, over and over again. The specific element or class that a particular name denotes must depend on the context in which that

name is uttered. In natural language there are three different kinds of context which we call *lexical, structural,* and *semantic.* The lexical context refers to the entire body of text in which the word is embedded. For example the noun *set* has distinct meanings if appearing in a mathematical treatise or an article about tennis. A word's structural context is its usage within the sentence or statement. The word *set* used as a verb is distinct from either of the two noun meanings above. More subtle distinctions may be derived from an understanding of the overall meaning of the sentence—this is the semantic context. Consider the two different meanings of "She set the vase on the table" and "He sets a high value on neatness". Programming languages have traditionally used both lexical and structural contexts to disambiguate the meanings of names. In section 2, we will extend the familiar concept of *lexical scope* from programming languages in order to provide a lexical context for name disambiguation in persistent environments.

In any language words must somehow be bound to their meanings. The role of dictionaries as a persistent mechanism for communally binding words to their meanings is well documented. But dictionaries are most important for binding the meanings of common nouns; only a few, extremely well known proper nouns are ever recorded here. Names, or proper nouns, are bound to the elements they denote by a variety of means—by telephone books, by voter registration lists, by gazetteers, by class year books. In section 3, we will describe the dictionary in more detail. At the same time we will show that the binding of class names (or common nouns) should be somewhat different from the binding of element names (or proper nouns, or unique identifiers).

Words change over time. New words are coined to denote new concepts and new instances. Some words actually change their meaning. And other words simply disappear from current usage. These phenomena are most difficult to model; and yet they are of crucial importance if one is to maintain a persistent name space. Sections 4 and 5 look at several important issues surrounding the dynamic denotation of words.

Finally, in section 6, we address the issue of synonyms—two distinct words which ave the same, or nearly the same, meaning.

2. Name Space Hierarchy

The meaning of words and names depends on their context. This is evident in natural language; meanings depend upon one's dialect, geographical location, economic position, educational level, and vocational speciality [Pei49]. In computer science, the importance of context has been most clearly formalized. Most readers will be familiar with a block structured language, such as Pascal or Algol, in which two identical words (variable identifiers), in separate procedures, can have completely different semantic meaning. With nested blocks, the semantic meaning of a name may, or may not, be inherited from the block at a lexically higher level. In task structured languages, such as Modula or Ada, words (and their meanings) can be explicitly *imported* from, and *exported* to other tasks.

There are several alternative ways of structuring a space of commonly known and understood *persistent* names. The structure can be quite chaotic with clusters of common understandings associated with different individuals in different situations, as it is in natural language. Or there can be just a single set of commonly understood words, as the reserved words of a programming language; their form and usage are completely determined. Of the many variations between these two extremes we have chosen to create an essentially hierarchical structure on our persistent name space. The levels of the hierarchy are

SYSTEM

GROUP or <groupid>

USER or <userid>

LOCAL

At each level, there is a dictionary containing the definition of every ADAMS word which is valid at that level in the hierarchy. Of course, SYSTEM, GROUP, and USER dictionaries are themselves persistent. A LOCAL dictionary (which might be called a symbol table) exists only for the duration of its associated process. Normally, when a word is encountered in an ADAMS

statement its meaning is first sought in the LOCAL dictionary, then the USER, GROUP, and SYSTEM dictionaries in that order.

Names and terms defined at the SYSTEM level have a universal meaning that is understood in common by all processes within the system. One would expect such core computer science words as "RELATION", "SCHEMA", "FILE", "RECORD", and "PROCESS", together with the primitive ADAMS *reserved* words of "ELEMENT", "SET", "ATTRIBUTE", "MAP", and "CO_DOMAIN" to be defined at this highest level. While it is expected that a number of commonly understood core classes (or types) will be defined at this level, there should be very few, if any, globally known objects or instances. Persistent object instances will tend to be owned by individual users or communally owned by small groups of users. (System devices, particularly if they are treated as files, might be one notable exception.)

The word "GROUP" used in an ADAMS statement denotes "my group". The definitions of all names shared by a group of users are found in that group's dictionary. In addition to "my group's" dictionary, any process may also make use of definitions occurring in the dictionary of another group whose identity, <groupid>, it knows. A process can *import* names.

The word "USER" refers to the persistent name space of a particular user. Although a user can easily create his own classes and data types, the entries in a USER dictionary will be, for the most part, the names of persistent objects and instances. This is quite analogous to file names in a user's directory. Object instances in another user's name space may be available by the import mechanism <userid> only if that user has provided export permission.

Finally, the term "LOCAL" denotes that space of non-persistent run time identifiers which are local to an executing process.

2.1. Name Paths

A *path* in the dictionary tree is a sequence of sub-dictionaries. The sequences <*user_3, group_b, system*> and <*proc_y, user_6, group_c*> in Figure 1 are paths. A *name path* is the path

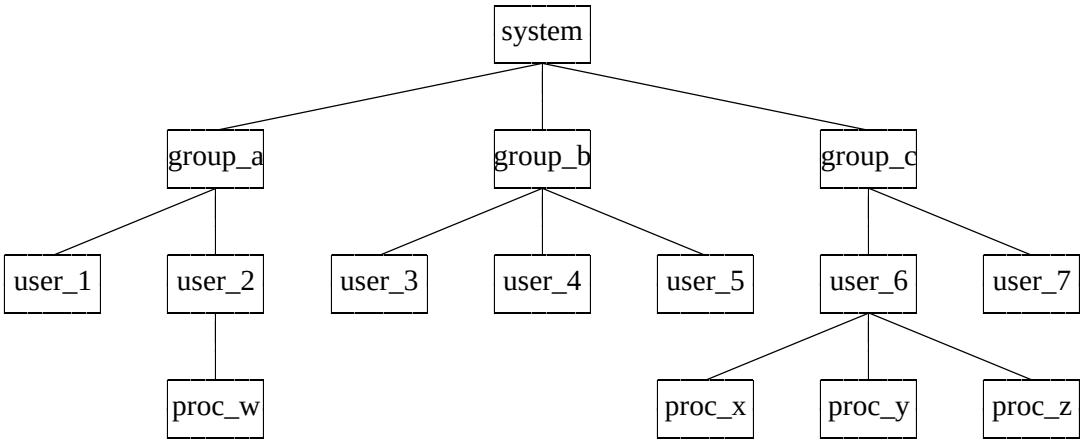


Figure 1
Hierarchically Structured Name Space

of dictionaries that any process could access in the course of resolving the meaning of a name. For example, the resolution of ADAMS names in process "proc_w" could involve searching along the name path

"proc_w"	(local)
"user_2"	(persistent)
"group_a"	"
"system"	"

We say a path beginning in a local dictionary is *active* with respect to a name, if there exists a process, which need not be currently executing, or another ADAMS element that references that name.

2.2. Examples of Scope Usage in ADAMS Statements

The structure of the name space hierarchy may become a little more clear if we look at some sample ADAMS statements. In ADAMS, we use a "class" concept to denote a general class of objects. The use of "class" in this context has been popularized by "Smalltalk-80" [CoM84, GoR83, 81]. However, many readers may feel more comfortable if they substitute the synonym *data type* for *class.* Specific objects are named instantiations of the class. The basic class definition statement has the form

<class_name> **is a** <class_designator> [<definition>][**with scope** <scope_id>].

Here, a new <class_name> is being defined in terms of an existing <class_designator> subject to any restrictions and/or extensions which will be imposed by the optional <definition> clause. If the optional <scope_id> is omitted, the new <class_name> will be LOCAL, by default. The basic object instantiation statement has the form

<instance_name> **belongs to** <class_name> [<instantiation_options>][**with scope** <scope_id>].

An instance of an object with <instance_name> belonging to the general class of objects with <class_name> is created and recorded in the appropriate dictionary.

Consider, for example, the following two ADAMS statements which we may assume occur in the text of "proc_z" in Figure 1 so that the name path is <*proc_z, user_6, group_c, system*>.

SCHEMA **is a** SET **of** ATTRIBUTE **elements, with scope** USER

faculty_attr **belongs to** SCHEMA **consisting of** { name, age, dept }

(For pedagogical reasons, we normally denote classes of ADAMS objects in upper case, specific instances in lower case, and reserved words in bold face. None of these are constraints of the language itself.)

Here a local definition of the schema concept has been created, in which "SCHEMA" denotes a set of attributes (as is customary in the relational model). The second statement then creates a specific relation schema, called *faculty_attr,* which has the properties of this newly defined SCHEMA class. Subsequent statements in "proc_z" can now create one, or more, relations using *faculty_attr* as the schema. The new *SCHEMA* definition will be persistent, so other processes belonging to "user_6" can also re-use the definition by simply referencing it. The *faculty_attr* instance is not persistent; it will vanish when the process terminates.

Had the ADAMS **"is a"** statement redeclaring the class SCHEMA not occurred in the text of "proc_z" then the instantiation creating *faculty_attr* would have searched up its name path to locate the semantics of "SCHEMA". Presumably a more traditional schema, whose semantics

consists of just a set of attributes, exists as a SYSTEM definition. Whether or not SCHEMA had been redefined, the SYSTEM version could have been invoked by the statement

faculty_attr **belongs to SYSTEM** SCHEMA **consisting of** { name, age, dept }

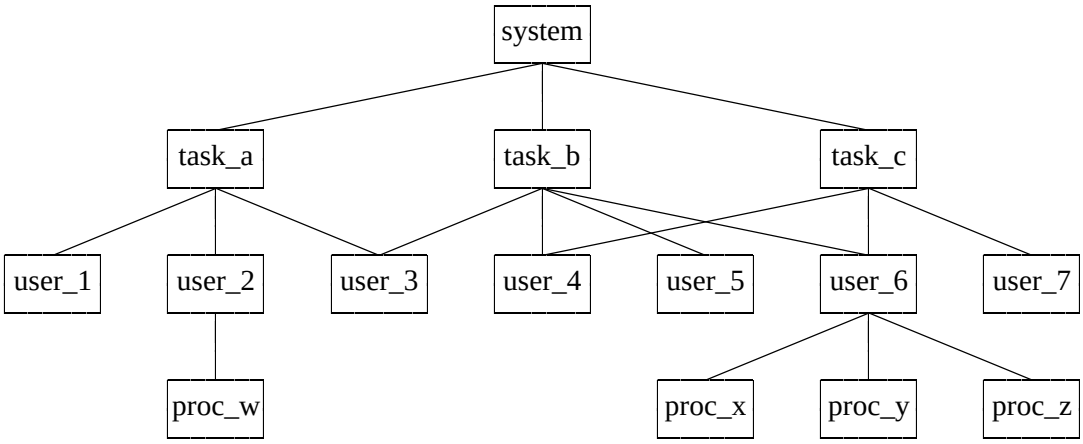
in which the <class_designator> *SYSTEM SCHEMA* would override the normal hierarchical scoping.

2.3. Reason for Choosing a Hierarchical Model

It is not apparent that the space of persistent names should be hierarchically structured. A SYSTEM dictionary of universal definitions which serves as the root of the hierarchy does make sense—even if there are only a few terms whose meaning can be commonly agreed on. A LOCAL name space of temporary identifiers used by a single process is also clearly required. And readily, there should be a USER dictionary that can record classes which have been defined in processes just of that user, as well as all persistent objects created by such processes. It is the GROUP level in the hierarchy which is questionable.

Following the UNIX tradition, the set of all users has been partitioned into disjoint sets called "groups". This is a reasonable structure for system administration; but it seems somewhat forced and unnatural as the structure of a name space. One would expect a task area to share common definitions; and one might expect a single user to participate in several different tasks with other users. A name space that was organized with respect to shared tasks would be an acyclic structure with a greatest element such as shown in Figure 2.

While a task structured name space might seem more natural than one that is hierarchically organized, the latter is much simpler. In a tree structured hierarchy there is a unique dictionary path from the ADAMS term occurring in a referencing process to the SYSTEM root. It is easy to resolve the meaning of names. Resolution of an ADAMS identifier in a task structured name space might entail searching along several paths in the acyclic network. For example, if we were using the structure of Figure 2 to resolve the semantics of a name in *proc_z* all of the tasks



Acyclic Task Structured Name Space
Figure 2

subsuming *user_6* (in this case just *task_b* and *task_c)* would have to be examined using a breadth first search. We decided to use the more rigid hierarchical name space in ADAMS simply because its implementation is just so much simpler. In practice, most of the rigidity can be circumvented because ADAMS permits direct reference to another name path in the tree by means of the explicit <group_id> and <user_id> scopes. It remains an open question as to whether this is a wise decision.

3. Dictionary Implementation

The ADAMS Dictionary is itself an ADAMS structure, and the underlying ADAMS operations are used to create and update the Dictionary.

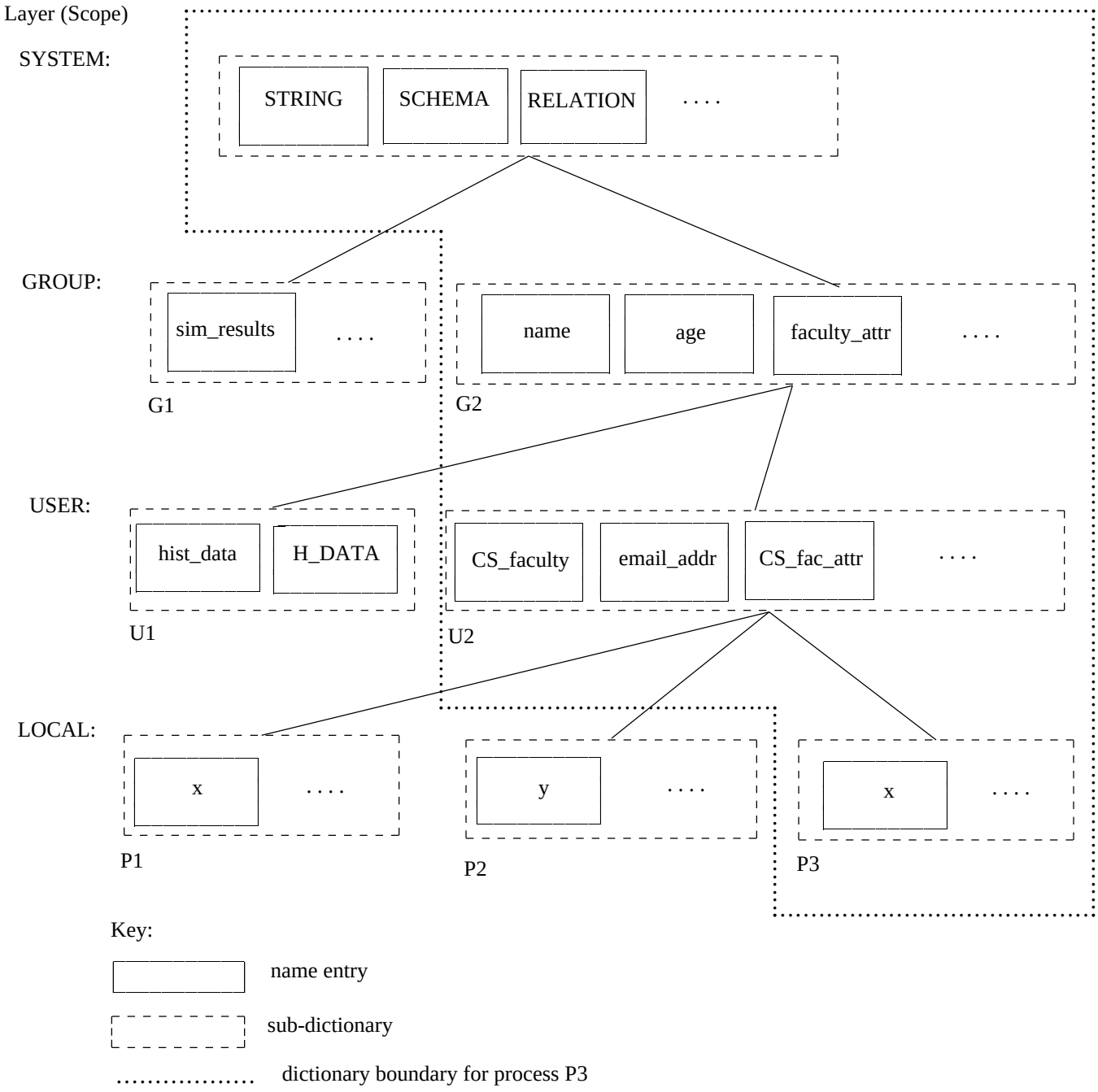
3.1. The Distributed ADAMS Dictionary Structure

A dictionary is a mechanism for looking up the meaning of words. That is precisely the purpose of the ADAMS Dictionary. Except instead of one monolithic tome, the ADAMS Dictionary is *logically distributed* into at least four scopes or levels. (Note: these levels may, or may not, be physically distributed.) The present concept describes a structure of four levels or scopes, three of which may contain numerous sub-dictionaries. Here the term *dictionary* (small d) refers to those sub-dictionaries which are referenced by a single process, or program, i.e., within a single name path; the *Dictionary* (large d) refers to the union of all dictionaries. We use *sub-dictionary* to denote that portion of a *dictionary* residing at one lexical level.

Figure 3 provides an example of a portion of a typical Dictionary. Logically, there is only one SYSTEM sub-dictionary, containing all terms defined for system-wide use. (In practice, we may have several replicated copies for efficiency.) There may be an arbitrary number of GROUP, USER, and LOCAL sub-dictionaries, forming a virtual tree, as described below.

3.2. Name Paths

Each USER and GROUP sub-dictionary has a unique internal identifier, used within the dictionary entries of its "children" (connected nodes, one level lower) to indicate the parent node. E.g., all USER-level entries contain the identifier of the GROUP-level sub-dictionary to which they belong. Concatenation of the sequence of these parent identifiers, beginning with a dictionary entry at any level and moving up the tree to the SYSTEM level, constitutes a path of sub-dictionaries which is the *dictionary* for that lexical level (and above). We call this path of sub-dictionaries a *name path* because all names must be resolved with the dictionaries of this path.



An Example Dictionary
Figure 3.

See Figure 3 for a simple example. Normally, an ADAMS statement will reference the name of

an entity without specifying a scope designation, as in:

faculty_attr **belongs to** SCHEMA **consisting of** {name, age, dept}

In this ADAMS statement the instance name *faculty_attr* is being defined. That definition is associated with the name *faculty_attr* in the local (temporary) dictionary by default. The name "SCHEMA" is referenced as a part of the definition of *faculty_attr;* the system will search for SCHEMA in order to use information from its dictionary entry to build a part of the *faculty_attr* entry. The local sub-dictionary is searched first, containing all currently active, temporary (non-persistent) names. If the entry for the name SCHEMA is not found here, the system determines the current user id, representing the "parent" user for the current local dictionary, and searches that USER-level sub-dictionary. If not found there, the search proceeds up the path to the parent GROUP sub-dictionary and then to the SYSTEM sub-dictionary until the name is found, or determined to be absent. A similar search process is employed to find the semantic meanings of *name, age,* and *dept.*

However, the user process may specify at which level of the dictionary to begin the search. With the ADAMS statement

faculty_attr **belongs to SYSTEM** SCHEMA **consisting of** { name, age, dept }

only the SYSTEM sub-dictionary will be searched for the "SCHEMA" entry..

Normal ADAMS usage allows duplicate names for different entities, as long as they have distinct scopes, that is, they occur in separate sub-dictionaries. Limiting name resolution searches to sub-dictionaries along the process’s name path protects against accessing the wrong entity. By providing for explicit scope designation in its referencing syntax, ADAMS also permits duplicate names in the same dictionary path, provided the names in any single sub-dictionary are distinct.

The ADAMS syntax allows reference to entries in other USER or GROUP dictionaries, as in:

use user_5 database **of user_5 RELATION**

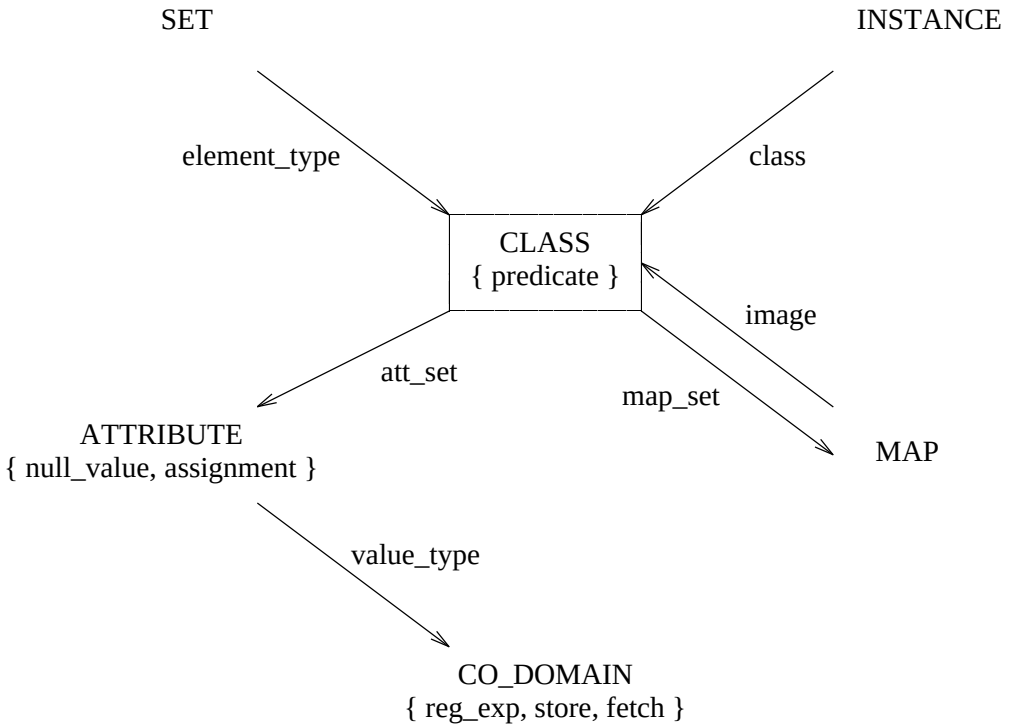
where the instance *database* belonging to *user_5* is accessed. Note that its definition will be found in user_5's dictionary. This should not be a frequent occurrence; if the name is used commonly by processes in other name paths, then it should probably be moved up one or more levels.

3.3. Individual Entries

Each named entity is defined by characteristics stored in its dictionary entry. Entities of different types require different information; six different dictionary entries have been defined at present: set, class, attribute, map, codomain, and instance. All ADAMS names fall into one of these six types.

Since the Dictionary itself is an ADAMS database, each of these entries can be defined and created by standard ADAMS statements. Each of the six constitutes a separate CLASS. Since classes are characterized (in part) by the attributes and maps defined for them, each *dictionary entry class* has a set of attributes defined for it, and may also have one or more maps that relate it to another entry class. Using the ADAMS system to "bootstrap" the ADAMS Dictionary itself causes a slight problem of nomenclature. To describe the Dictionary structure with a minimum of confusion, we will use three terms, which we apply only to internal dictionary entries. First, we will call the six individual entry classes *categories*, so that the word "class" will only refer to classes other than the dictionary entry classes. Instead of referring to the internal attributes defined for each of these entry categories as "attributes", we will call them *fields.* Thus the term "attributes" is reserved for the external, programmer-defined attributes associated with a class, while "fields" refers to the internal attributes associated with a given category of dictionary entry.

In addition to a set of fields, five of the six categories have relationships mapping elements of the category to those of another entry category. We choose the term *dependencies* to refer to the internal maps defined on entry categories, while "maps" is reserved for external, programmer-defined maps.



Entry/Dependency Schematic

Figure 4.

A schematic of interrelationships between the six categories of entries is shown in Figure 4. Properties of each entry category are listed under its name; arrows indicate dependencies, and point to the image category for the dependency. As an example, class name entries have dependencies (by means of *att_set* and *map_set)* to *sets* of attribute and map entries, which are two defining characteristics of a class. As would be expected, the class entry is the most common and has dependencies to or from all other entry categories, with the exception of codomain entries. Every dictionary entry, regardless of category, has two fixed fields not shown on the diagram—its *name*, on which any search is directed, and a set of *references,* which will be described later.

The schematic of figure 4 is generally illustrative of the dependencies; but it ignores many details. We give our first detailed definition of the Dictionary using C-like syntax.

```
struct
      {                          /* dict(ionary)_entry */
      char          *entry_name;
      E_TYPE        entry_type;
      REF_SET       *references
      SYN_SET       *synonyms
      union
            {
            CO_DOM_ENTRY  co_dom;
            CLASS_ENTRY   class;
            INST_ENTRY    instance;
            MAP_ENTRY     map;
            ATTR_ENTRY    attribute;
            }
      } DICT_ENTRY

struct
      {                          /* Body of Co_Domain entry */
      char          *reg_expression;
      int()         *mem_test;
      int()         *fetch_conv;
      int()         *store_conv;
      } CO_DOM_ENTRY;

struct
      {                          /* Body of Class/Set entry */
      DICT_ENTRY    *parent;
      char          *predicate;
      int()         *pred_test;
      DICT_ENTRY    *element_type;      /* only classes of type SET */
      ASSOC_SET     *att_set;           /* associated attributes    */
      ASSOC_SET     *map_set;           /* associated maps          */
      } CLASS_ENTRY;

struct
      {                          /* Body of Attribute entry */
      DICT_ENTRY    *co_domain;
      char          *null_value;
      ASSIGN_TYPE   assignment;
      *char()       *computed;
      } ATTR_ENTRY;

struct
      {                          /* Body of Map entry */
      DICT_ENTRY    *image;
      } MAP_ENTRY;

struct
      {                          /* Body of Instance entry */
      DICT_ENTRY    *inst_class;
      ELEM_ID       *id;
      } INST_ENTRY;
```

Dictionary Layout in C
Figure 5.

Note that every dictionary entry (DICT_ENTRY) has an *entry_name* field, by which all entries are accessed; an *entry_type* tag field, identifying the kind of dictionary entry, a *references* field,

denoting all other dictionary entries and/or processes which reference this entry, and a *synonyms* field, denoting all synonymous entries. The remaining body of the entry belongs to one of five basic types: *co_dom, class, map, attribute,* or *instance.*

If the entry denotes a co_domain, then its body contains a field, *reg_expression,* denoting the regular expression by which the co_domain has been defined. Such an expression is of little practical use and is maintained only for display purposes. At run time a membership test predicate, *mem_test,* is used to determine whether an assigned value actually belongs to the co_domain. (Note: it may be necessary to also provide "equality" and "ordering" test processes.) The procedures *fetch_conv* and *store_conv* are used to convert the co_domain value, which is by definition a string, into a computational form appropriate for the particular hardware when it is fetched from storage—or conversely placed in storage.

Many of the dictionary entries will define classes. There are four basic types of classes, CLASS, SET, ATTRIBUTE, and MAP. The class type is also denoted by *entry_type.* Whenever a class is defined in terms of a pre-existing class, it automatically inherits the properties of that class. That pre-existing class is denoted by the *parent* field. This link can be followed to access any inherited properties or characteristics. Additional properties of the current class are defined by a *predicate* expression. The integer function *pred_test* is invoked during execution to test whether or not newly created, or modified, instances actually satisfy the predicate restrictions for membership in the class.

If the ADAMS class is a SET, then the *element_type* field is used to verify that newly inserted elements are of the correct type, and that various set operations are legitimate. This field is used to enforce homogeneity in all ADAMS sets. Attribute functions and map functions can be declared on any ADAMS class. Sets of such attributes and maps are associated with the class by

means of the *a_set* and *m_set* fields.

Co-domain, attribute, and map entries are all used in ADAMS in the definition of element classes. A class is a kind of generic data object, or a data type. Of course they must be named, and their names must be recorded is the Dictionary so that they can be referenced by various processes. An *instance* is an actual data object belonging to some class. Every ADAMS instance, or element, is uniquely identified by a symbolic *id,* which is sufficient to access the element in secondary storage. But it need not be named. Most instances are not named. For example, given a set of elements, it is customary to name the set as a whole (which is an instance), but not to name any of its individual constituent elements. Thus for any named instance, the only dictionary fields that are required, in addition to the standard fields of *name, entry_type, references,* etc., are the instance *class,* to access a complete description of the instance type, and *id,* to access the data object itself.

As noted earlier, the ADAMS Dictionary is persistent. Consequently, it structure can be described within the ADAMS syntax itself. Given below, in figure 6, is such a definition. Readers, unfamiliar with this syntax should refer to [PSF88]. Note that both this definition and the C definition given above are incomplete; a number of "low-level" types and co_domains are left undefined.

The "union" of structures in C has always been an awkward construct, as has the "variant" record in Pascal syntax. They are mechanisms for asserting that a class of structures are "almost" the same, but not quite. Storage is allocated for the largest possible structure. The actual internal structure must be designated by some other "testable" variable to insure correct run time behavior. A "union" concept is needed because a dictionary entry can be either a class entry, a co_domain entry, or an instance entry, and each has somewhat different internal structure. ADAMS has no provision for "variant" elements. The elements of a set, such as the entries in a dictionary, must be homogeneous. Instead, the variant portions of the these entry bodies are

```
entry_type      belongs to ATTRIBUTE, with image E_TYPE
reg_expression  belongs to ATTRIBUTE, with image STRING
mem_test        belongs to ATTRIBUTE, with image BOOLEAN_PROC
fetch_conv      belongs to ATTRIBUTE, with image BOOLEAN_PROC
store_conv      belongs to ATTRIBUTE, with image BOOLEAN_PROC
predicate       belongs to ATTRIBUTE, with image STRING
pred_test       belongs to ATTRIBUTE, with image BOOLEAN_PROC
id              belongs to ATTRIBUTE, with image ELEM_ID
null_value      belongs to ATTRIBUTE, with image STRING
assignment      belongs to ATTRIBUTE, with image ASSIGN_TYPE
computed        belongs to ATTRIBUTE, with image STRING_PROC

references      belongs to MAP, with image REF_SET
synonyms        belongs to MAP, with image SYN_SET

ASSOCIATION     is a CLASS, having fields = { synonym, map/att_set }
ASSOC_SET       is a SET, of ASSOCIATION elements

a_set           belongs to MAP, with image ASSOC_SET
m_set           belongs to MAP, with image ASSOC_SET

DICT_ENTRY      is a CLASS, forward

co_domain       belongs to MAP, with image DICT_ENTRY
image           belongs to MAP, with image DICT_ENTRY
element_type    belongs to MAP, with image DICT_ENTRY
parent          belongs to MAP, with image DICT_ENTRY
inst_class      belongs to MAP, with image DICT_ENTRY

ATTR_ENTRY      is a CLASS, having dependencies = { co_domain },
                        having fields = { null_value, assignment, computed }

MAP_ENTRY       is a CLASS, having dependencies = { image }

INST_ENTRY      is a CLASS, having dependencies = { inst_class, attr_def, map_def },
                        having fields = { id }

CLASS_ENTRY     is a CLASS, having fields = { predicate, pred_test },
                        having dependencies = { parent, element_type, a_set, m_set }

CO_DO_ENTRY     is a CLASS, having fields = { reg_expression, mem_test, fetch_conv, store_conv }

co_dom          belongs to MAP, with image CO_DO_ENTRY
class           belongs to MAP, with image CLASS_ENTRY
instance        belongs to MAP, with image INST_ENTRY
map             belongs to MAP, with image MAP_ENTRY
attribute       belongs to MAP, with image ATTR_ENTRY

DICT_ENTRY      is a CLASS, having fields = { entry_name, entry_type },
                        having dependencies = { references, synonyms, co_dom, class, map, attribute, instance }
```

Definition of the Dictionary in ADAMS syntax
Figure 6

declared to be distinct classes, here INST_ENTRY, CLASS_ENTRY, MAP_ENTRY, ATTR_ENTRY, and CO_DOM_ENTRY. A DICT_ENTRY then has three associated maps,

instance, class, map, attribute, and *co_dom* whose image will be an instance of one of the classes respectively. Depending on the *entry_type* only one of the maps will be defined—the other two will be *null* maps. So, for example if *x* denotes a dictionary entry of *entry_type* "co-domain", the ADAMS expression *x.co_dom.reg_expression* will denote the regular expression (a string) which defines the co_domain.

4. Migration of Names

ADAMS is designed to support dynamic development of systems. Its name space must therefore be dynamic. New kinds of classes can be defined and entered into the system. Procedures can then be written to operate on instances of these classes—to test their utility, to test their adequacy. The usual approach to such innovation is to experiment with new constructs at an individual USER level. If the new construct seems to be robust and useful, it may be moved up to GROUP status, where it can be tested by a larger community of users. Similarly, particularly useful GROUP constructs and/or objects can be installed at a SYSTEM level. Such movement is called upward *migration.* The upwards migration of a class or instance name is accomplished by a statement of the form

rescope [<current_scope>] <name> **to** <new_scope>

Although it occurs less often, names of limited use can migrate downwards.

Readily, no name (character string) can be moved to a level where the same string exists as a name with a different semantic definition. It might change the behavior of processes referring to that name. But absence of the name from the target dictionary is not by itself sufficient to ensure that a new entry or an upward (or downward) migration will not have unwanted side effects. Addition, deletion, or movement of a name in a hierarchically structured space can unintentionally mask, or unmask, other definitions of the same name.

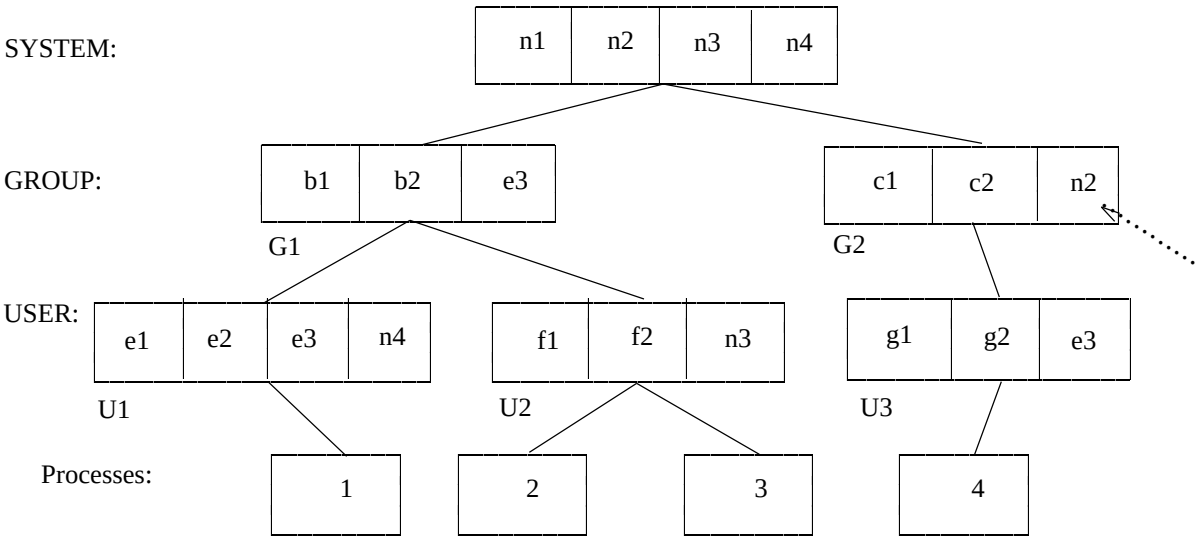
4.1. Name Insertion and Upward Migration

A newly defined name may be inserted into a sub-dictionary provided (1) a duplicate name does not already exist in that sub-dictionary, and (2) inserting the name at that level does not interfere with other name paths. For example, if a process expects to find a given name at the SYSTEM level, insertion of an entry for an identically-named entity into the GROUP sub-dictionary along the name path causes the search to find this new entry rather than the intended

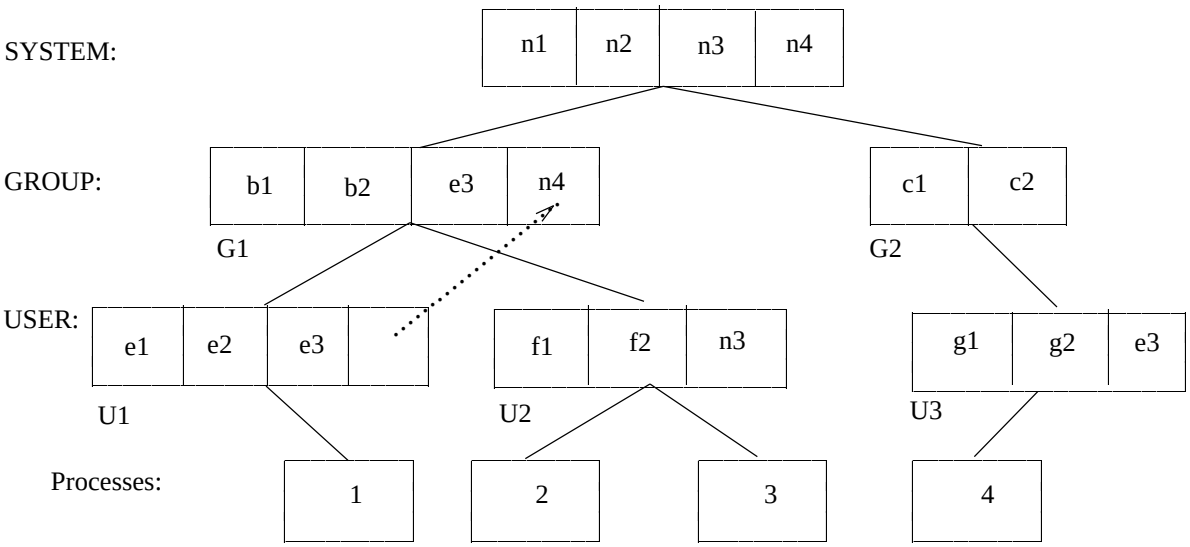
top-level one as in figure 7.

Moving an entry upward in a name path can cause the same problems as insertion. Raising an entity one level will not of itself prevent a search from locating the entry; it will merely be found one level higher. However, the move can mask higher entities in other name paths. In figure 8, moving the name *n4* up one level masks the SYSTEM level definition of *n4* from the processes of U2.

It is database instances that tend to be most dynamic; it easy to envision them being created, used, and released by processes. Consequently, we tend to think in terms of issues relating to instances being accessed along active name paths by processes. But, it is important to remember that any persistent element, or instance, must belong to a specific CLASS—thus the instance in turn references a CLASS definition. Therefore that CLASS definition cannot be migrated in such a way that it becomes masked in the name path through the instance entry.



Insertion of entry *n* 2 into G2 masks SYSTEM level *n* 2.
Figure 7.

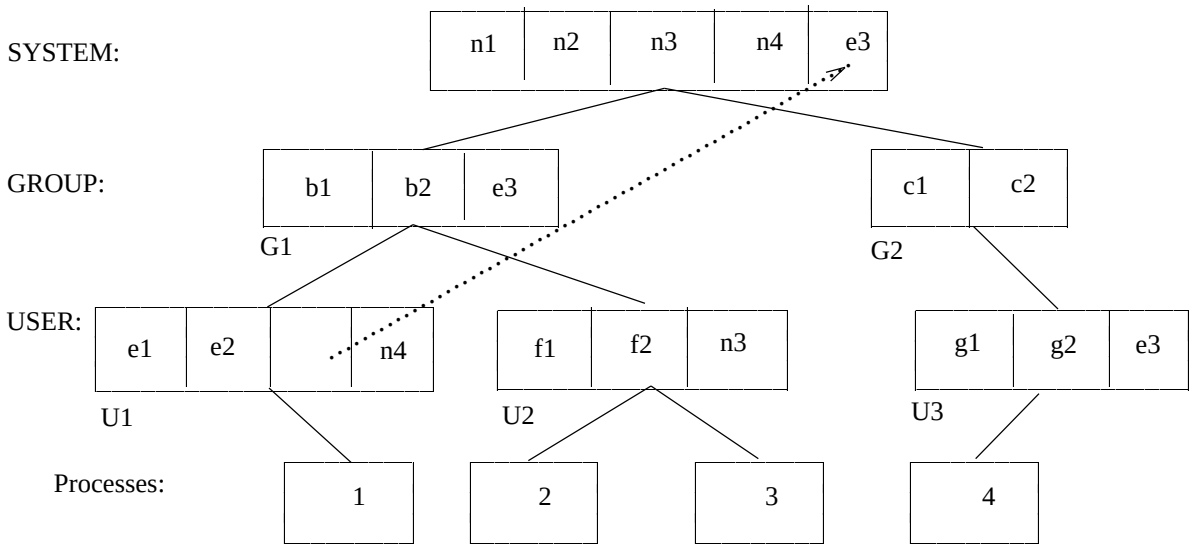


Moving *n* 4 up from U1 to G1 masks SYSTEM level *n* 4 for processes 2 and 3.
Figure 8.

In figure 9, moving the name *e3* up two levels unmasks the different definition of *e3* in the G1 sub-dictionary. Consequently, the processes of U1 may not access the definitions, or instances, they had been using. Moving *e3* into a sub-dictionary "behind" the occurrence in the U3 sub-dictionary should cause no problems. But observe that incorrect deletion of *e3* in the U3 sub-dictionary would unmask the SYSTEM level *e3.*

4.2. Removal of Dictionary Entries

Data removal, or deletion, is always more difficult than data insertion. A user, or process, may specify that a class name, co_domain name, or instance_name should be removed from a sub_directory because it is to be moved to another sub-directory, or it is no longer needed and should be deleted altogether from persistent storage. Can it, in fact, be removed? Let us consider a few sample cases.



Moving *e* 3 from U1 up two levels to SYSTEM, unmasks *e* 3 in G1
and also causes a potential problem for any subsequent deletion of *e* 3 in U3.
Figure 9.

Suppose, first, that we seek to delete a class name altogether. If there still exist instances of this class, then readily the class entry can not actually be deleted from the dictionary. Similarly, if the class is referenced as the image of a MAP, or as the parent of another CLASS, it must be kept—even though its original creator may believe it has out lived it usefulness. We call these *internal* references because they represent cross references within the persistent ADAMS name space; not references by external processes. We could largely ignore the issue of internal cross references during the discussion of name insertion because, of course, no name can be cross referenced until it has been installed into the Dictionary. It was sufficient to consider only the issue of masking other name paths.

A CO_DOMAIN entry can be referenced by an ATTRIBUTE class definition. A CLASS entry can be referenced as the

class of an instance,
parent of another class,

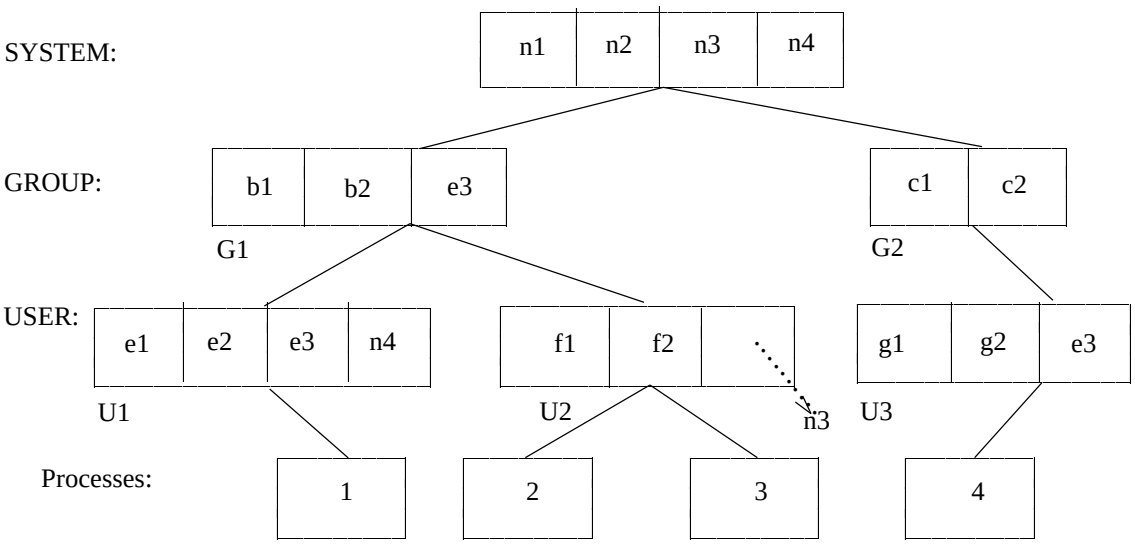
image of a map class, and
element_type of a set class.

Attribute and map instances are referenced in class definitions. All of these internal cross-reference dependencies, which were summarized in Figure 4, must be tested and found to be empty before removal can take place.

We saw that name insertion, or upward migration, could have surprising side effects in the form of masking, or unmasking, other names in the space. And, just as insertion of an identical name can mask a desired name entry, deletion of an entry can uncover access to an identical but unwanted name at a higher level. See, for example, Figure 10.

4.3. Summary

The preceding examples can be condensed into the following three rules which govern the definition, migration, and deletion of names in the name space.



Deletion of *n* 3 from U2 unmasks *n* 3 at SYSTEM level
Figure 10.

(1) No name may be entered into a target dictionary level if the target dictionary is on an active name path to a different definition of the same name.

(2) No name may be deleted from a dictionary which is on an active name path to the occurrence of the name in a different process.

(3) Names must be (conceptually) moved one level at a time.

Readily, in order to support a dynamic name space, one must have a way of determining which name paths to any given name in the space are active.

5. Active Name Paths

In section 2.1, we said that a name path was active if there existed a process which referenced that name along the path. And, because the name space is permanent, we emphasized that the process need not be currently executing. Given the concept of internal cross-referencing introduced in the preceding section, the active name path concept must be extended to include those name that are indirectly referenced by active processes.

One would expect such an extension to complicate the determination of active name paths. In fact it does not. Keeping track of internal cross-references is much easier than determining active references in existing, but dormant, processes. Part of the ease derives from the nature of internal cross-referencing itself, part derives from the rule that any term used in an ADAMS declaration must be found in the same sub-dictionary, or in a higher level dictionary. Search for these internal cross-references is relatively easy.

The problem of determining whether or not a persistent object is referenced by another object or process is a common theme in computer science. It appears, for example, in list processing, in storage management, and in shared memory maintenance. List and graph processing using pointers requires some mechanism to determine when a pointer may be changed or deleted without impacting on other parts of the program. Some storage allocation methods require a means of determining when a block of storage can no longer be used and thus can be safely freed. Shared memory cells require some means of knowing when it is safe to update the cell and maintain consistency among the various users' viewpoints. Various methods have been developed to deal with these situations.

Garbage collection is a useful, automatic way to remove items that can no longer be referenced, and hence used. The method appears as an integral component of LISP interpreters, to deallocate storage for LISP functions or lists no longer active [WiH81]. It may also provide a means of deallocating heap storage no longer referenced by active processes. Garbage collection

depends on two factors; a means of marking the objects to be retained or deleted, and a means of inspecting the parts of the system that *might* use the objects. In both of the examples mentioned (LISP and general heap clean-up), this second set of objects to be inspected consists of a limited set of *active* processes. The storage units that might be referenced by these processes are not persistent, and the resultant interactions are accordingly limited in number. In contrast, ADAMS dictionary entries are persistent, and the entities referencing them are programs, which may be executing or dormant. Inspection of all programs that might require a given entry in essence means inspecting the object code symbol tables of all stored programs. An approach to determining active name paths which makes the same assumptions as garbage collection would be clearly difficult to implement.

A number of researchers have designed *lock* mechanisms that in effect permit various degrees of access to items (nodes of a tree, pages of a file) and allow update only when the lock is exclusive (as defined by the system). Carla Ellis [Ell80], expanding on work by Bayer and Schkolnick [BaS77], defined three different locks, one for reading a node, two for writing. One of the "write" locks allowed only one process access at a time; the other two locks kept counts of processes using the lock on a given object at any one time. All such schemes insure consistency by requiring at least one exclusive lock for updating an object. The use of "lock counts" is essentially the same as the maintenance of reference counts, described below, with an exclusive lock having a count of one.

Maintenance of pointers is often supported by *reference counts.* UNIX directory entries keep track of links between files in this fashion, with no file being completely deleted from the system until all link references have been removed, indicating that no directory in the hierarchy contains a pointer to the file. An early, but generally effective, mechanism to ensure the safe removal of subordinate list structures has appeared in [Wei63] and [Pfa77]. It requires a record of the number of referencing super-lists.

5.1. Methods of Determining Active Name Paths

There appear to be two basic methods of determining whether a specific object is useful, that is might be referenced in the future, or whether it can be deleted from permanent storage. One involves the use of reference counters, the other makes use of a modified garbage collection method. A third method, which will not be further explored in this report, might assume the existence of "write once, read often" memory in which nothing would ever be actually deleted from permanent storage.

It is possible to record with each dictionary entry the number of times that that entry is referenced by other named constructs and/or entries, which by virtue of being named, also occur as dictionary entries. These are cross-references within the name space.

CLASS entries can be referenced as
(1) a parent class of a restricted CLASS,
(2) the element class of a SET,
(3) an image class of a MAP, and
(4) the class of a named INSTANCE.

CO_DOMAIN entries can be referenced only as
(1) the co-domain of an ATTRIBUTE.

INSTANCE entries can be referenced as
(1) in the sets of MAPS and/or ATTRIBUTES associated with a CLASS.

Whenever a new dictionary entry is created, the reference counters of all entries that are cross-referenced in the course of its definition would be incremented. Whenever an existing dictionary entry is deleted—its reference counter would have to be zero—the reference counters of all cross-referenced entries would be decremented.

More commonly, of course, instance entries will be referenced in executable ADAMS statements appearing within program segments. Also class entries can be referenced by *unnamed* instances within the class. Indeed, it is quite possible to have a class of ADAMS elements in which no individual instance is ever named; the elements of a named set constitute an excellent

example. These two considerations would appear to generate severe problems.

Processes containing ADAMS statements must run in some environment. If this environment were controllable by ADAMS, as it is in LISP programs, then a solution to the first problem, references within executable code, could be envisioned. The object version of each process would have to have a prefix that enumerated the *named* ADAMS elements and constructs appearing within the code. (Recall, we are only maintaining a "name" space; ADAMS variables can be ignored.) Every time a compiler created such an object process, it would have to access the dictionary and increment the appropriate reference counters. If the object process were removed, say by an operating command such as *rm proc_a*, the operating system would have to access the dictionary and decrement all of the reference counters indicated in the prefix. It is possible to envision such a totally controlled environment in the future; it is impossible to envision such changes to the general operating system environment in the present. An option might be to associate with each instance name in the dictionary, the date it was last referenced. Then a heuristic procedure could periodically examine the entire dictionary and mark some instance entries that have been unreferenced for an extended time as "presumed dead". These instances would then be deleted, and associated reference counters decremented.

Unnamed instances can reference dictionary entries; they must reference at minimum the CLASS to which they belong. Since permanent instances can only be created, and deleted, by ADAMS commands this situation can, in theory, be handled by ADAMS. Whenever a persistent element is created, whether named or not, all appropriate reference counters could be incremented. They could be decremented when it is deleted from permanent storage. Such a procedure would be tedious and inefficient. More important, it is unnecessary. There can be no instances which are unreachable from some named instance. Therefore it is only necessary to maintain cross references between named ADAMS constituents, that is cross references within the dictionary itself. We need only maintain a name space, not an entire storage space.

6. Synonymy

Two words are said to be *synonymous* if they mean the same thing—that is they can be used interchangeably. In natural languages, one seldom has exact synonymy; usually the words mean *nearly* the same thing and can be used interchangeably in most contexts. But invariably there are subtle differences and nuances that will make one of the synonyms more appropriate than another. Consequently, in natural languages one can speak of close and distant synonymy, where these terms roughly denote the contextual scopes over which the words are equivalent. In English, the classic definition of synonymy is Roget's Thesaurus.

The issue of synonymy is an important one in ADAMS, as indeed it must be in any system which maintains a persistent name space. Communication between people, or processes, which employ a large number of distinct, but synonymous, terms for shared concepts is difficult. In effect, they are using different languages, or at least different dialects of the same language. There is a real communication advantage to detecting synonymous words, noting the synonymy, and where possible replacing the synonyms with a common word. For example, the terms *relation, tuple,* and *attribute* are virtual synonyms to *file, record,* and *field* in most database applications. But simply because different words are used it is often difficult to compare results found in the relational literature with that found in the literature of file processing. And procedures written in one genre are seldom translated into the other. There would be distinct benefits to bringing these two dialects into congruence. In the context of ADAMS, it would permit increased sharing of data constructs and communication between procedures. It would also reduce the size and clutter in our dictionaries.

While there are some decided advantages to replacing a plethora of synonyms with common expressions, there are also some disadvantages. On one hand, there may be subtle differences between apparent synonyms. In the preceding example, while the relational terms are virtually synonymous to the file processing terms, the former are somewhat more abstract while the

latter tend to be implementation specific. Both sets of concepts are usually implemented identically; but they need not be. A tuple need not be represented as a single record with an ordered sequence of fields. Further, in an experimental environment, which ADAMS has been designed to support, meanings tend to change frequently. My private definition of a *relation* may currently be synonymous with a commonly accepted definition, but may change tomorrow. I may want to keep my meaning distinct.

In our development of ADAMS we have tried to develop formalisms and mechanisms which would support various interpretations of synonymy and to enforce what seems to be a reasonable standard—we generally enforce a strong control over synonymy at the SYSTEM level, where these terms are assumed to be widely accepted, a weaker control over synonymy at the GROUP level, and no control over the use of synonymous terms at the USER or LOCAL levels.

Still, there is much that we do not understand and much more to be investigated.

6.1. Definition of Synonymy

Synonymy is an equivalence concept. In ADAMS, two terms are said to be *strongly synonymous* if all terms occurring in their definitions are identical. For example, if the terms *student_record* and *s_tuple* were defined by the ADAMS statements

```
student_record is a CLASS
        having { student_number, name, gpa }

s_tuple is a CLASS
        having { name, student_number, gpa }
```

then *student_record* and *s_tuple* are strongly synonymous. They are defined in terms of the same set of identical attributes. Note that the order in which the attributes have been enumerated is different. This is deliberate. ADAMS is defined in terms of sets, not sequences, so these definitions are identical.

It is easy to verify that strong synonymy is an equivalence relation.

Two ADAMS co-domains are also said to be strongly synonymous if they define the same regular set. Other ADAMS terms are said to be *weakly synonymous* or, just *synonymous*, if all terms in their definitions are synonymous. (Note, identical terms are clearly synonymous.) For example, if

name **belongs to** ATTRIBUTE
with image STRING, **value is** assigned

s_name **belongs to** ATTRIBUTE
with image STRING, **value is** assigned

are the respective definitions of *name* and *s_name*, then they are strongly synonymous. If now *s_tuple* were to be redefined by

s_tuple **is a** CLASS
having { s_name, student_number, gpa }

then *student_record* and *s_tuple* would be weakly synonymous because the attributes *name* and *s_name* are only synonyms, not identical.

Again, it is easily verified that weak synonymy is also an equivalence relation. And strong synonymy implies weak synonymy.

It is important to note that synonymy in ADAMS refers to structural, or syntactic, synonymy. The representations of the objects denoted by the terms (either dictionary entries or instances) must be congruent. This is distinct from what might be called semantic synonymy. If we were to define two attributes

age **belongs to** ATTRIBUTE
with image INTEGER_3, **value is assigned**

and

length **belongs to** ATTRIBUTE
with image INTEGER_3, **value is assigned**

both of which are functions into to the same co_domain, which in this example is all 3-digit

integers, then structurally they are strongly synonymous. But we would not wish to treat them as synonymous in a semantic sense. The distinction between structural and semantic synonymy will create problems in creating effective synonymy enforcement protocols.

6.2. Synonymy Implementation

The implementation of synonymy in a persistent name space presents three important sub-issues. They are the (1) detection of synonymy, (2) the representation of synonymy, and (3) the enforcement of appropriate synonymy protocols.

The representation of synonymy is the easiest to handle. Since both forms of synonymy are true equivalence relations, it is sufficient to simply represent as a set the equivalence class (of other terms) that are synonyms. Moreover, operationally there is no need to maintain a distinction between strong and weak synonymy so a single set will suffice. We use the *synonyms* field of a dictionary entry to do this. This field is then used to (1) alert a user to the presence of synonymous terms, and (2) detect other occurrences of synonymy.

The detection of synonymy is more difficult and time consuming. At the time of definition (term binding), all other entries in the same sub-dictionary must be examined for possible synonymy. A simple screen based on surface appearance will eliminate most candidates; it will also immediately detect strong synonymy. Discovery of weak synonymy in the few remaining candidates will require testing for synonymy between pairs of defining terms, including a possibly exhaustive comparison of all permutations of associated sets of attributes and maps.

There are three features of the ADAMS language and its persistent name space which aid in the discovery of synonymy. First, simply recording sets of all synonymous terms in the dictionary entry eliminates the need for a recursive descent search. Second, all ADAMS constructs must be defined in terms of names on the same name path, that is in the same sub-dictionary or higher level dictionaries. Given its hierarchical organization, this reduces the scope of the name space that must be examined. Finally, ADAMS requires that all co-domains must be regular sets

defined by regular expressions. Thus, in theory (c.f. [HoU79]) the test for whether or not two co-domains are synonymous is decidable. (In practice, ADAMS does not test for synonymy between co-domains so only strongly synonymous attribute functions will be discovered.)

Finally, we address the issue of synonymy enforcement protocols. In ADAMS, we decided to test for synonymy only at the GROUP and SYSTEM levels of the name space, not at the USER or LOCAL levels. This decision was based on considerations of efficiency—there are far fewer terms in the dictionary at these levels; and because the value of inter-user and inter-process communication is far more apparent at this level. But having discovered synonymy, what actions should be taken? We would like to develop protocols which would alert the user to synonymy at the GROUP level, and which would prohibit synonymy altogether at the SYSTEM level. If ADAMS were a purely interactive system, it would be easy to control the use of synonymy through such protocols. In the former case, the user could be interactively notified of pre-existing synonymous terms and asked whether one of them could be used instead. In the latter case, the user would be notified of the other synonym and instructed that it must be used. But ADAMS is not intended to be used with only interactive processes. We do not know how to enforce any control over synonymy in such a non-interactive environment.

7. References

[BaS77] R. Bayer and M. Schkolnick, Concurrency of Operations on B-Trees., *Acta Inf. 9*(1977), 1-21.

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

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

[Ell80] C. S. Ellis, Concurrent Search and Insertion in 2-3 Trees, *Acta Inf. 14*(1980), 63-86.

[GoR83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison Wesley, Reading, MA, 1983.

[HoU79] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[KhC86] S. N. Khoshafian and G. P. Copeland, Object Identity, *OOPSLA '86, Conf. Proc.*, Sep. 1986, 406-416.

[MSe86] D. Maier, J. Stein and et.al., Development of an Object-Oriented DBMS, *OOPSLA '86, Conf. Proc.*, Sep. 1986, 472-482.

[Pei49] M. Pei, *The Story of Language*, J.B. Lippincott, Philadelphia, 1949.

[Pfa77] J. L. Pfaltz, *Computer Data Structures*, McGraw-Hill, Feb. 1977.

[PSF87] J. L. Pfaltz, S. H. Son and J. C. French, Basic Database Concepts in the ADAMS Language Interface for Process Service, IPC TR-87-001, Institute for Parallel Computation, Univ. of Virginia, Nov. 1987.

[PSF88] J. L. Pfaltz, S. H. Son and J. C. French, The ADAMS Interface Language, *Proc. 3th Conf. on Hypercube Concurrent Computers and Applications*, Pasadena, CA, Jan. 1988, 1382-1389.

[Wei63] J. Weizenbaum, Symmetric List Processor, *Comm. ACM 6*,10 (Oct. 1963), 524-536.

[WiH81] P. H. Winston and B. K. P. Horn, *LISP*, Addison Wesley, Reading, MA, 1981.

[81] The Smalltalk-80 System, *BYTE*, Aug. 1981, 36-48.

Table of Contents