Implementation of the ADAMS Database System

John L. Pfaltz, James C French
Andrew Grimshaw, Sang H. Son
Paul Baron, Stanley Janet
Yi lin, Lindsay Loyd,
Rod McElrath

IPC-TR-89-010
December 4, 1989

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 and DOE Grant #DE-FG05-88ER25063.

IPC Technical Report 89-010

Implementation of the ADAMS Database System

John L. Pfaltz, James C. French
Andrew Grimshaw, Sang H. Son
Paul Baron, Stanley Janet
Yi Lin, Lindsay Loyd,
Rod McElrath

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract:

ADAMS provides a mechanism for applications programs, written in many languages, to define and access common persistent databases. The basic constructs are *element, class, set, map, attribute,* and *codomain.* From these the user may define new data structures and new data classes belonging to a semantic hierarchy that supports multiple inheritance.

This report describes the prototype implementation of ADAMS. Several features of interest include: the way a shared persistent name space has been captured in a dictionary structure, the implementation of a preprocessor which embeds runtime C++ proceses into host application programs written in standard C, the implementation of an object based uniqueid for all ADAMS elements, a single functional O-tree operator to represent sets, maps, and attributes, and a low-level storage operator to distribute data items to multiple storage devices.

This research was supported in part by DOE Grant #DE-FG05-88ER25063 and JPL Contract #957721

1. Overview, Philosophy

The goal of ADAMS (Advanced DAta Management System) is to create a large virtual persistent memory which processes can use to store and access items of data in much the same way that programming languages now allow the storage and access of data in volatile memory. Approaches to accomplish this kind of data management have been proposed in Persistent Algol [BuA86] and ODE [AgG89]. Three differences which characterize the ADAMS approach are that, (1) the virtual persistent memory can be shared between distinct users if desired; (2) it is not language specific—the programmer may code in any of the standard programming languages, such as C, Pascal, Ada, or Fortran; and (3) it is specifically designed for, although not limited to, parallel applications.

The fundamental concept of the ADAMS approach is that of an *element.* An element is not an item of data. Rather it should be viewed as a place holder with which data values can be associated. The concept of an *element* is not unlike that of a storage location in traditional memory management, or an object in object-object oriented languages. Like objects (or volatile memory locations) elements must belong to a *class*, that is the nature of the element must be well-defined. These classes are programmer defined and can be hierarchical. Elements are uniquely identified. Regarding elements as if they were objects can be helpful to those readers who are familiar with object-oriented programming; indeed, the runtime ADAMS system has been implemented in C++ to take advantage of many of its object-oriented constructs. But, in reality elements are *not* objects.

Other basic ADAMS constructs are: *sets*, (one frequently deals with sets of elements, rather than the individual elements themselves); *attributes*, (which associate data values with an element); and *maps* (which associate elements with each other). Finally, there is the concept of a *codomain*, (which conceptually defines a class of data values) and a *subscript pool*, (which enumerates a particular sequence of codomain values that can be used to subscript identifiers). The syntax of ADAMS, which governs how these preceding constructs are formally related to each other, is completely described in [PFG89]. The reader is referred to this report for these syntactic details. In this report we concentrate solely on our mechanisms for implementing these constructs, and of the ADAMS language as a whole.

In Figure 1.1 we illustrate our conception of ADAMS. One, or more, processes are running on distinct processors (which we have illustrated as a tightly coupled 8 node hypercube, although any distributed set of processors could be used). ADAMS statements are embedded in the code of these processes and provide for the definition, access, and storage of items in virtual persistent storage, which is represented in the figure by the two disk units. From the programmers point of view, this code is interacting directly with this persistent storage via the ADAMS interface. In fact, the interface is accomplished by the ADAMS run-time system which we have represented as the small 4 node hypercube in to the right in the figure. User processes make requests to ADAMS procedures of the run-time system (in the figure $proc_3$ is schematically connected to $adams_2$ and $proc_6$ is connected to $adams_0$) which are directly connected to the disk representations. In the figure, the ADAMS run-time system is represented as existing on a separate hypercube. This might be a desirable configuration. Alternatively, certain portions of the ADAMS run-time system might be executing as co-processes on the individual user's nodes in order to minimize message transmission.

A different schematic is shown in Figure 1.2. In this figure, we indicate that a "host" program (here *host* denotes the language such as C or Pascal, not a host node as in hypercube terminology) is first translated by the ADAMS *preprocessor* into pure host language code without embedded ADAMS statements. Such source code with embedded ADAMS statements must be designated with a *.src* suffix, as in <*filename*>*.src* The preprocessed source program without ADAMS statements will suffixed with *.c* or *.p* or *.f* depending on the host language used. For example, preprocessing a C program *testprog.src* will generate a new file *testprog.c*. This version

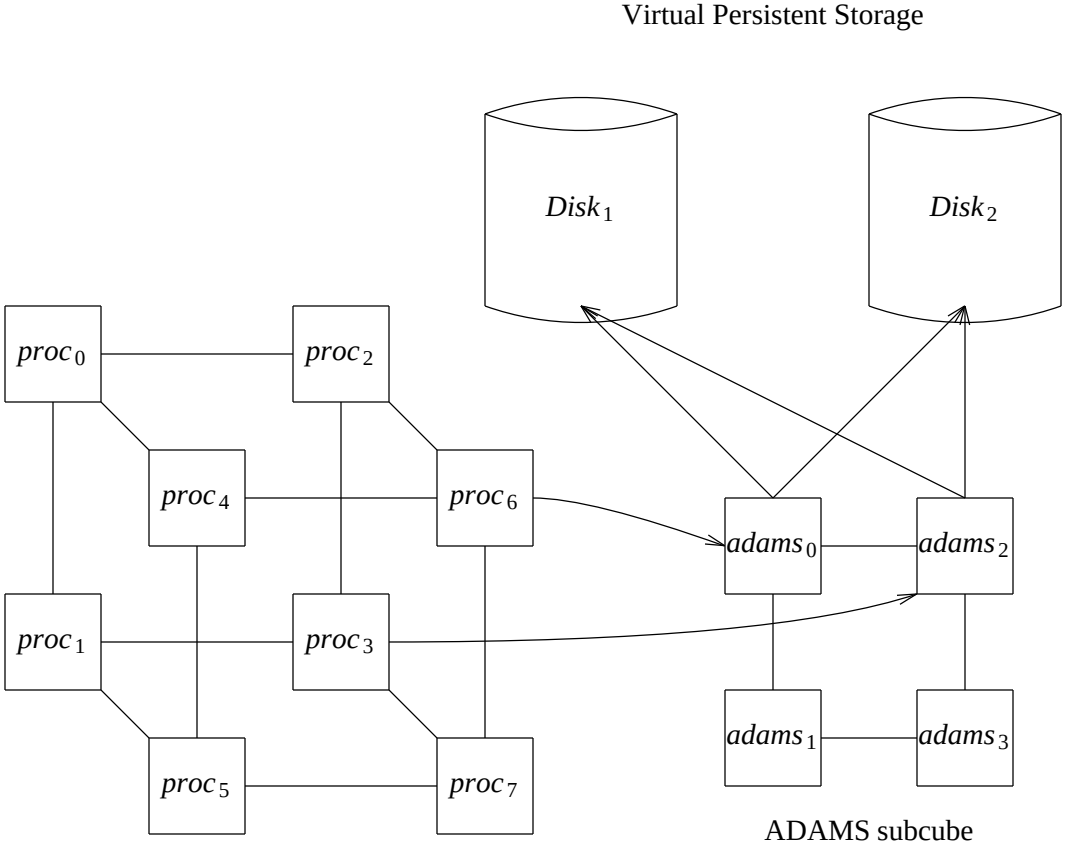

Figure 1.1
Virtual ADAMS Environment

of the source program is then compiled in the usual manner by the host language compiler, as shown in the figure, to obtain an executable program.

The executable program, when run, invokes the runtime ADAMS system as necessary. This runtime system invokes the persistent dictionary and various attribute, map, and set representations. Although discussed separately in sections 7, 8, and 9 because originally we envisioned the implementation of maps, sets, and attributes as distinct system components, these have be unified by a single functional concept pfaltz french implementing subscripted scientific 1990 , pfaltz functional approach aggregation SIGMOD 1990 .] which is symbolized by the right-most box in the figure. The box lablelled "runtime ADAMS system" thus serves primarily to interface these functional representations with the persistent dictionary and the users application program as indicated. Its central routines are covered primarily in section 7.

An important characteristic of Figure 1.2 is the dashed line through the *UNIQUE_ID construct* box. All interface with persistent items stored by the ADAMS database manager is in terms of these unique id's. Every ADAMS *element* is assigned a *unique_id* on creation (instantiation). It never changes for the lifetime of the element. These unique id's, which are invisible to the user (programmer), are the only means by which ADAMS references its elements. All literal identifiers (names) are bound to unique id's, either at preprocessing time, or when absolutely required, at run time. The closest analogy is to the binding of variable and procedure

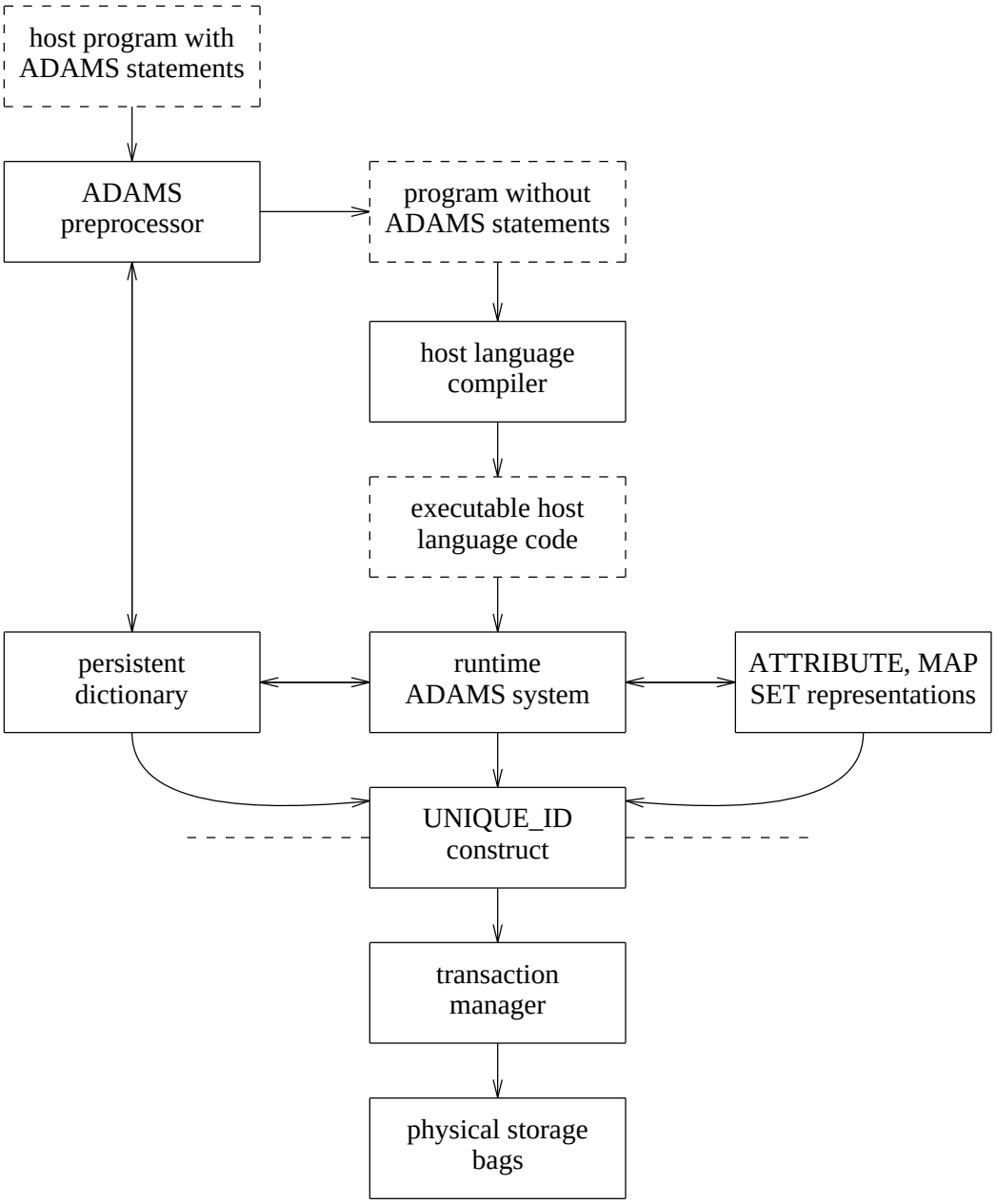

Figure 1.2
Overall ADAMS Structure

identifiers (names) of a traditional programming language to relocatable storage locations at compile time. The *transaction manager* uses these unique id's to access items in physical storage. We call such physical storage a *bag* to avoid confusion with a more traditional file concept. A *bag* of ADAMS items may reside on any storage device.

2. The Run-Time System

2.1. Directory Layout

The directory layouts for both the prototype version on *atlas*, a VAX 8600, and *ice*, an INTEL iPSC-2, are described below.

The initial prototype version has been implemented on atlas. It resides in the common directory */at0/adams*. This directory consists of the 8 subdirectories listed below.

```
TEST            ( A prototypical environment )

bin

include

lib

obj

src             ( ADAMS source code )
        cc.d
        dict.d          ( dictionary source                 )
        parse.d         ( preprocessor source               )
        sets.d          ( set representation source         )
        uids.d          ( unique_id, ADAMS element source   )

Server  (bag code)

Client      "
```

2.1.1. TEST

The TEST directory represents a prototypical environment—that is, it includes the ADAMS dictionary, and "user" directories which oontains ADAMS source (.src) programs to compile and run.

The structure of TEST is as follows:

```
adamsdict       ( common, shared ADAMS dictionary )

pkb4h           ( This individual's test programs )

jlp                     "       "

 .                      "       "
 .                      "       "
 .                      "       "
```

In effect this is a small completely contained user environment. More realistically, the *adamsdict* would reside in some known system location.

The *adamsdict* has the structure shown below. Note that currently all users operate under the task *t123* by default.

```
adamsdict   (common, shared adams dictionary )
        cl      ( SYSTEM classes   )
        co      (    "      codomains )
        hd      (    "      headers   )
        in      (    "      instances )
        sp      (    "      subscript pools )
        t123    ( Task 123  -- current default )
                cl      ( TASK classes   )
                co      (   "     codomains )
                hd      (   "     headers   )
                in      (   "     instances )
                sp      (   "     subscript pools )
                u567    ( imaginary user )
                        cl      ( USER classes   )
                        co      (   "     codomains )
                        hd      (   "     headers   )
                        in      (   "     instances )
                        sp      (   "     subscript pools )

                u853    ( pkb4h -- Paul's user dictionary )
                        cl
                        co
                        hd
                        in
                        sp
                  .
                  .
                  .
        t500    ( Task 500 )
          .
          .
          .
```

2.1.2. include

The *include* directory contains the following header files.

C headers:

bag_cc.h	Declaration of bag operations; used by the actual bag implementation.
dict_cc.h	C version of dictionary function declarations; included by user programs (include statement inserted by preprocessor).
indexglue.h	C "glue" to C++ runtime code for tables and sets. Included by user programs (by preprocessor).
indexman.h	C calls to the index manager, a runtime structure that maintains open indexes and sets. Included by user programs (by preprocessor).
symbol.h	Declaration of table used by preprocessor to look up symbols and their values.
uid_cc.h	C version of _A_uid declaration; included by user programs (include statement inserted by preprocessor).

C++ headers:

bag.h C++ interface to bags.
Included by: uidmgr.h

defs.h Commonly used defines and constants (e.g., _A_BOOLEAN)
Included by: attr.c, fwdattr.c, fwdmap.c,
index.c, indexman.c, indexglue.c,
invatt.c, invmap.c, key.c,
map.c, set.c, uid.c

dict.h Dictionary functions.
Classes: _A_METHOD
_A_M_ITERATOR
_A_CLAS
_A_C_ITERATOR
_A_ASSO_SET
_A_A_ITERATOR
_A_SUBVALUE
_A_S_ITERATOR
_A_ENTRYDEF
_A_CO_DEF
_A_CLASS_DEF
_A_SUB_DEF
_A_INST_DEF
_A_LIST_CHUNK
_A_REF_REC
_A_SUBDICT
Included by: All dictionary source files.

extern.h Commonly used objects. (So far, just _A_NULLUID and _A_NULLKEY: used for comparison of _A_uid and _A_key return values.)
Included by: attr.c, fwdattr.c, fwdmap.c,
index.c, indexman.c, indexglue.c,
invatt.c, invmap.c, key.c,
map.c, set.c, uid.c

hdrs.h Simplified, independent versions of _A_uid and some uidmgr structures. Needed by code that initializes a bag server and sets up the uniqueid tables. (Not used by runtime system.)

indexes.h Basic classes of tables (currently implemented using collections).
Includes: key.h
Classes: _A_index
_A_fwd_map (from _A_index)
_A_fwd_attr (from _A_index)
_A_set (from _A_index)
_A_inv_map (from _A_index)
_A_inv_attr (from _A_index)
Included by: attr.c, fwdattr.c, fwdmap.c,
index.c, indexman.c, indexglue.c,
invatt.c, invmap.c, key.c,
map.c, set.c, uid.c

key.h Class declarations for search keys.
Classes: _A_key

	Included by:	indexes.h
mapattr.h	Maps and generalized attributes.	
	Classes:	_A_attr (uses _A_fwd_attr, _A_inv_attr)
		_A_map (uses _A_fwd_map, _A_inv_map)
	Included by:	attr.c, indexglue.c, indexman.c, map.c
uid.h	Uniqueid class declaration.	
	Classes:	_A_uid
	Included by:	indexman.c, indexglue.c
		Runtime system source programs
		for tables, sets, and dictionary.
uidmgr.h	Uniqueid tables.	
	Includes:	bag.h
	Classes:	_A_uidmgr (from _A_index)
	Included by:	uid.c, uidmgr.c

2.2. System Interface

In the prototype version we have a number of paths hardwired. Two, in particular, you must include in your operating environment by modifying your *.login* or *.profile* file. Add

```
BAGID=/at0/adams/Server
BAGSOCKET=/at0/adams/Server/_SOCKET_

export BAGID BAGSOCKET
```

You should also add */at0/adams/bin* and */at0/adams/TEST/bin* to your PATH variable.

Since, ostensibly, the TEST directory represents the complete prototype environment, we expect you to create your own directory under TEST and run test cases there. However, this is not required.

Finally, you must create a user dictionary, by executing the command

```
create <unix_id>
```

If you do not know your <unix_id>, grep the */etc/passwd* file using your login id.

3. Preprocessing, Translating ADAMS Statements

Here we include just a summary overview. More complete details can be found in [Bar89].

Assume that you have created a source program consisting of mixed C code and ADAMS statements in a file <prog_name>.src. The *.src* suffix is required, since the ADAMS preprocessor will look for it. The command

adams <prog_name>

will translate this source program and produce a new source program consisting of only C statements called <prog_name>.c. Notice that this has the customary *.c* suffix.

To compile the resulting C code in <prog_name>.c execute the command

cci <prog_name>

Note, *cci* is a shell script in */at0/adams/TEST/bin* (as is *adams*) which simply imvokes the standard *cc* compiler, but also includes the necessary glue routines to interface the C code generated by the ADAMS preprocessor with the C++ routines of the run-time system. *cci* will generate an executable file <prog_name> with no suffix. <prog_name> should now be executable.

Should the preprocessing step fail, check that you have created a user dictionary as described in section 2.3.1.

4. Persistent Names, the Dictionary

The basic way by which host language programs reference elements in the persistent data space managed by ADAMS is by "name". Attributes, maps, and classes are named. Many sets (which are analogous to files in more conventional systems) are named. Any ADAMS element may be named, including individual instantiations within sets. Of couse, many interesting data retrievals employ constructs other than names alone; for example, one may retrieve a set of elements satisfying some specified properties. But, still the concept of literal names used in a host language program to identify specific elements (or unique_id's) is central.

Since the elements so identified are persistent, the name space used to identify them must also be persistent. In this section, we explore the concept of the ADAMS name space, and the dictionary concept which is used to record information about certain kinds of named elements and to establish the correspondence between literal names and the unique_id of the element so identified.

4.1. Overview of the ADAMS Name Space

ADAMS has a persistent name space. All classes, codomains and subscript pools must be named. Instances may have names. This name space is organized in a hierarchical structure. The levels of the hierarchy are

SYSTEM
TASK
USER
LOCAL

Every user has his own persistent dictionary which contains all the names that are defined and used only by himself. This dictionary is at the USER level. The LOCAL level dictionary contains the names that are created and used only during the execution of a single program. When the program terminates, this dictionary no longer exists. Several users can form a task so that they can have common names shared only by themselves. All their common names are in the TASK level dictionary which is the parent of their USER dictionaries. Note that every user

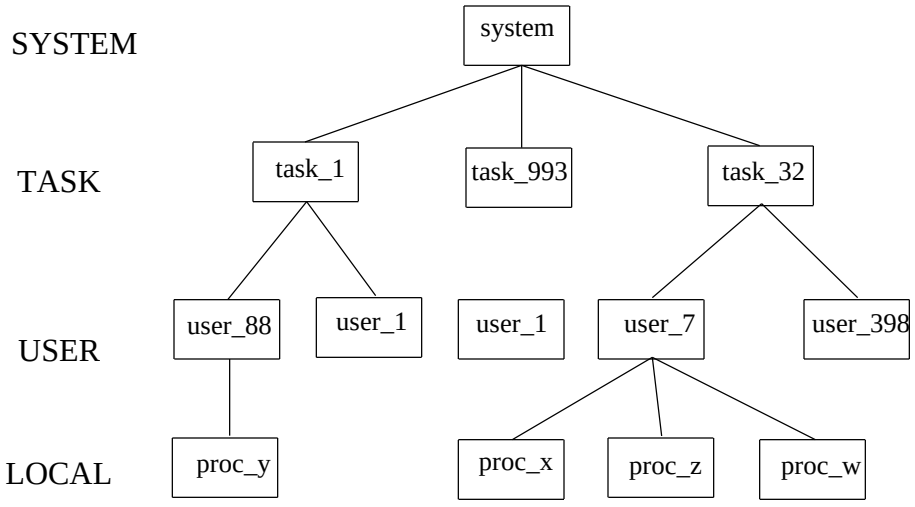

Figure 4.1 Hierarchically Structured Name Space

must belong to at least one task. A user can be in more than one task at the same time. In such a case he will have different USER dictionaries for each of the tasks. In Figure 4.1, *user_1* is in both *task_1* and *task_993*. From the dictionary point of view, these user dictionaries are not related to each other although they are used by the same human user. The SYSTEM level dictionary contains all the names that can be used by everyone. There is only one such dictionary for the whole system.

A human user of ADAMS always works in an environment in which the user ID and task ID are specified. This means he can only "see" the SYSTEM dictionary, the TASK dictionary of that task ID, the USER dictionary of his user ID under that task and the LOCAL dictionary for the program he is running.

When an ADAMS name is created, its scope is specified so that it can be entered into the correct level. Normally, when a name is encountered in an ADAMS statement, its meaning is first sought in the LOCAL dictionary, then the USER, TASK, and SYSTEM dictionaries in that order. We use the word *sub-dictionary* to denote a particular dictionary of some level. For example, the SYSTEM dictionary itself is a sub-dictionary. The dictionary for a particular task or a particular user of a task is a sub-dictionary.

The ADAMS name space is dynamic. New names will be created. Old names may be deleted or rescoped from one level to another level. In doing these, some consistency rules should be followed. The fundamental rules are:

a) A name can be inserted or rescoped to a sub-dictionary, only if it does not mask another entry with the same name.

   For example, if there is a name *a_class* in the SYSTEM sub-dictionary and it is the super-class of an instance *a_inst* in the USER sub-dictionary of task_a, user_88, we cannot enter another class with the same name *a_class* into the TASK sub-dictionary of task_a because this new class will mask the one in the SYSTEM sub-dictionary.

b) A name can be erased from a sub-dictionary, only if it is not referenced by any other dictionary entries or ADAMS elements.

c) A name can be rescoped away from a sub-dictionary, only if it will still be visible to all the names that reference it

d) A name can be rescoped upward only if all the other names that it references will still be visible to it; i.e. it will not reference any names that are below the target level.

4.2. What Does the Dictionary Do?

The basic function of the ADAMS dictionary is to provide a definition for a given name. Every name in the dictionary has an entry and a unique ID. A dictionary entry can be considered as a data structure. There are four types of entries:

codomain entries,
class entries,
subscript pool entries, and
instance entries.

For example, a codomain entry contains the definition of the named codomain. We separate each entry into three parts: the name part, which contains the name string and unique ID of that entry, the definition part, which contains the definition of that name, and the reference record part, which keeps the reference counts for all those tasks and users that are referencing it. (This part is not implemented yet.)

The definition formats of these four types of entries are as follows:

codomain:

validating method (name string of the validating method)
fetch method (name string of the fetch method)
store method (name string of the store method)
other methods (a list of name strings of other methods)
undefined value (undefined value string)
unknown value (unknown value string)

class:

super classes (a list of super class unique ID's)
element class (set element class, only for a set class)
image (image unique ID, only for attribute class or map class)
associated attribute sets (a list of set instance unique ID's, possibly with synonyms)
associated map sets (same as above)
boolean method (name string of the boolean method for restriction)

subscript pool:

codomain (unique ID of the value codomain)
values (a list of subscript value strings)

instance:

super classes (a list of super class unique ID's)

What the dictionary basically does is to provide a mechanism so that when a new name is defined, a dictionary entry can be filled out and inserted into the dictionary; when the definition of a name is needed, that corresponding entry can be accessed to retrieve its definition. Of course, we also provide the mechanism for deletion and rescoping.

The dictionary is used both at pre-processing time and at run time. At pre-processing time, the pre-processor gets the definitions of all literal names it encounters, performs static type checking and replaces them with unique ID's in its generated code. It performs the same operations on the dictionary as at run time, but these operations are not actually performed on the persistent dictionary. No changes are written back to the persistent dictionary. It is only at run time that the dictionary is really changed. New entries are inserted and old ones may be rescoped or deleted.

Unlike all other sub-dictionaries, when the SYSTEM sub-dictionary is first created, it is not empty. There are six predefined classes in it, ie CLASS, ATTRIBUTE, MAP, SET, ATTRSET (attribute set), and MAPSET(map set).

4.3. Dictionary Interface and Glue Routines

The dictionary interface is composed of two types of data structures, a *dictionary entries* and an *iterators*, which the user (i.e. ADAMS routines such as the preprocessor, or runtime procedures) manipulates by a set of dictionary routines.

4.3.1. Dictionary Entry

The most important data structure in the dictionary interface is the dictionary entry structure. Since we have four different types of entries and since we want our dictionary interface routines to be polymorphic, the way we adopted is to define a generic entry structure which is solely used for interfacing purpose. This structure is called _A_ENTRYDEF. It is used in two ways:

(1) When a new name is defined, an _A_ENTRYDEF object is filled out with the definition of the name and then it is inserted into the dictionary.

(2) When the definition of a name is needed, that name is looked up and its definition is copied to an _A_ENTRYDEF object. Then different fields of the definition can be extracted.

Before an _A_ENTRYDEF object can be filled out with the definition of a new name, it must be first set to the correct entry type, ie. codomain, class, subscript pool or instance entry. When an _A_ENTRYDEF object is used to get a definition from the dictionary, it can contain anything before the look-up; the old stuff will be overwritten if the name is found. (If it is not found, the look-up routine will return some integer value indicating the failure. So the content of the object is irrelevant.)

Three distinct kinds of operations can be applied to _A_ENTRYDEF objects:

– setting the type of the entry
– filling out entry fields
– extracting data from entry fields

4.3.2. Iterators

Many fields of dictionary entries are *lists* of unique ID's or method names, such as lists of super classes and associated sets. To extract these fields from an _A_ENTRYDEF object, we use several kinds of iterators. An iterator can extract such a list from the entry object, then present the elements in the list one by one as the result of a "get_next" operation on it. We have four kinds of such iterators:

method iterator	*(_A_M_ITERATOR, for other-method list of codomain)*
class iterator	*(_A_C_ITERATOR, for super class list of class and instance)*
associated set iterator	*(_A_A_ITERATOR, for associated set list of class)*
subscript pool value iterator	*(_A_S_ITERATOR, for subscript pool value list)*

4.3.3. Dictionary Routines

So far we have introduced two kinds of data structures, i.e. _A_ENTRYDEF and iterator, that are used independently in the access of the dictionary. The major work is actually done by the dictionary routines in conjunction with these two data structures. All these dictionary routines are explained in the following summary of the dictionary interface. Note that only the pre-processor will make full use of the interface. In the run-time code generated by the pre-processor, only a sub-set of the interface is needed because some work is done only at pre-processing time.

Types and Constants:

```
enum  _A_BOOLEAN  { _A_FALSE, _A_TRUE };
enum  _A_ENTRYTYPE { _A_CO, _A_CLASS, _A_SUB, _A_INST, _A_HDR, _A_DEF,
                     _A_REF, _A_NTYPE };
const _A_DEFTYPE  _A_CODEF    = 0x00;     // COdomain DEFinition
const _A_DEFTYPE  _A_CLASSDEF = 0x01;     // CLASS DEFinition
const _A_DEFTYPE  _A_SUBDEF   = 0x02;     // SUBscript pool DEFinition
const _A_DEFTYPE  _A_INSTDEF  = 0x03;     // INSTance DEFinition
const _A_DEFTYPE  _A_NDEF     = 0x04;     // No DEFinition

const _A_D_LEVEL  _A_SYSTEM   = 3;
const _A_D_LEVEL  _A_TASK     = 2;
const _A_D_LEVEL  _A_USER     = 1;
const _A_D_LEVEL  _A_LOCAL    = 0;
const _A_D_LEVEL  _A_NLEVEL   = -1;

const _A_CLSTYPE  _A_SUPER    = 0x01;     // SUPER bit (SUBCLS or SUBSET)
```

```
const _A_CLSTYPE  _A_CLS      = 0x02;     // CLASS
const _A_CLSTYPE  _A_SUBCLS   = 0x03;     // SUBCLaSs
const _A_CLSTYPE  _A_MAP      = 0x04;     // MAP class
const _A_CLSTYPE  _A_ATTR     = 0x08;     // ATTRibute class
const _A_CLSTYPE  _A_SETCLS   = 0x10;     // SET CLaSs
const _A_CLSTYPE  _A_SUBSET   = 0x11;     // SUB-SET class
const _A_CLSTYPE  _A_ATTRSET  = 0x30;     // ATTRibute SET class
const _A_CLSTYPE  _A_MAPSET   = 0x50;     // MAP SET class
const _A_CLSTYPE  _A_NCTYPE   = 0x00;     // Not Class TYPE
```

Dictionary Entry Declarations:

```
class  _A_ENTRYDEF
      {
   public:
      _A_ENTRYDEF (_A_DEFTYPE d_type = _A_NDEF);
      ~_A_ENTRYDEF(){ clear(); }
      void  set_type (_A_DEFTYPE);
            // An object must be set to the proper type,
            // ie. _A_CO_DEF, _A_CLASS_DEF etc. before it can
            // be used to define a new definition

      void operator = (_A_ENTRYDEF&);
            // Assignment Operation

                              // CO-DOMAIN DEFINITION OPERATIONS

                              // The following functions add different kinds of
                              // methods to the definition. The parameters are
                              // character strings which are supposed to be the
                              // method name.
      void  add_validm (const char *valid_m);
      void  add_fetchm (const char *fetch_m);
      void  add_storem (const char *store_m);
      void  add_otherm (const char *other_m);

      void  add_udf (const char *udfstr);
            // Add the undefined value to the definition.
      void  add_ukn (const char *uknstr);
            // Add the unknown value to the definition.

                              // The following functions get the names of
                              // codomain methods.
      _A_BOOLEAN  get_validm (char *valid_m);
      _A_BOOLEAN  get_fetchm (char *fetch_m);
      _A_BOOLEAN  get_storem (char *store_m);
      void  get_otherm (_A_M_ITERATOR& iterator);

      _A_BOOLEAN  get_udf (char *udfstr);
            // Get the undefined value string.

      _A_BOOLEAN  get_ukn (char *uknstr);
            // Get the unknown value string.
```

```
                    // CLASS DEFINITION OPERATIONS

void  add_superclass (const char *c_name, _A_D_LEVEL scope);
      // Add super classes to a class definition.
      // 'Scope' is the scope of the class being defined.

void  add_elementclass (const char *classname, _A_D_LEVEL scope);
      // Add the element class to a set class definition.

void  add_image (const char *imagename, _A_D_LEVEL scope);
      // Add the image to an attribute or map class definition.

void  add_attrset (const char *setname, _A_uniqueid& setid,
             _A_D_LEVEL lv, const char *synonym = NULL);
      // Add associated attribute sets to a class definition.
      // 'setname': the name of the set if it has one
      // 'setid'  : the unique id of the set
      // 'lv'     : the scope of the class definition
      // 'synonym': the synonym of the associated set, if it
      //            has one

void  add_mapset (const char *setname, _A_uniqueid& setid,
            _A_D_LEVEL lv, const char *synonym = NULL);
      // Add associated map sets to a class definition.

void  add_booleanm (const char *boolean_m);
      // Add the boolean method name to the class definition.

_A_CLSTYPE  get_classtype ();
      // Get the type of the class, eg. a sub-class, an
      // attribute, a map, or a set.

void  get_superclass (_A_C_ITERATOR& iterator);
      // Get all the super classes of a class.

_A_uniqueid  get_elementclass ();
      // Get the set element class.

_A_uniqueid  get_image ();
      // Get the image uniqueid of an attribute or a map.

void  get_attrset (_A_A_ITERATOR& iterator);
      // Get all the associated attribute sets (unique id's).

void  get_mapset (_A_A_ITERATOR& iterator);
      // Get all the associated map sets (unique id's).

_A_BOOLEAN  get_booleanm (char *boolean_m);
      // Get the boolean method name.

                        // SUBSCRIPT POOL DEFINITION OPERATIONS

void  set_codomain (const char *co_name, _A_D_LEVEL scope);
      // Set the codomain of a subscript pool.

void  add_subvalue (const char *valstr);
```

```
      // Add a subscript pool value ( a character string).

      _A_uniqueid  get_codomain ();
            // Get the codomain id of a subscript pool.

      void  get_subvalue (_A_S_ITERATOR& iterator);
            // Get all the value strings of a subscript pool.

                              // INSTANCE DEFINITION OPERATIONS

      void  add_instclass (const char *class_name, _A_D_LEVEL scope);
            // Add a super class to the instance definition.

      void  get_instclass (_A_C_ITERATOR& iterator);
            // Get all the super classes of the instance
            // definition.
      } ;
```

Iterator Declarations:

```
      class  _A_M_ITERATOR          // _A_METHOD iterator
            {
         public:
            _A_M_ITERATOR ();
            _A_BOOLEAN next (char *);
            } ;

      class  _A_C_ITERATOR          // class iterator
            {
         public:
            _A_C_ITERATOR ();
            _A_BOOLEAN next (_A_uniqueid&);
            } ;

      class  _A_A_ITERATOR          // _A_ASSO_SETS iterator
            {
         public:
            _A_A_ITERATOR ();
            _A_BOOLEAN next (_A_uniqueid&, char * = NULL);
            };

      class  _A_S_ITERATOR          // _A_SUBVALUE iterator
            {
         public:
            _A_S_ITERATOR ();
            _A_BOOLEAN next (char *);
            } ;
```

To show how the iterators work, in the following example we extract all the super classes of an instance using an _A_C_ITERATOR object.

```
_A_ENTRYDEF   def;
_A_C_ITERATOR iter;
_A_uniqueid   id;
...
```

```
def.get_instclass(iter);      // Set the iterator object to the super
                              // class list of the instance whose
                              // definition is already in "def" after
                              // a look-up.

while (iter.next(id))         // Extract unique ID's of super classes
                              // into "id" one after another
                              // until 0 (_A_FALSE) is returned.
      {
            // A super class has been extracted and its unique ID is
            // the current value of "id".
      ...
      }
      // Now every super class has been extracted.
```

Dictionary Functions:

```
int  _A_attach_dict (unsigned tid, unsigned uid, char *file_name,
                     char *directory, _A_BOOLEAN pre_pro)
      // Attach the SYSTEM, "tid"-TASK, "uid"-USER subdictionaries
      // to this process.
      // "file_name" is usually the name of the user program file.
      // "directory" is the dictionary directory.
      // "pre_processor" indicates if this function is called by
      // the pre_processor or by the user program.


_A_BOOLEAN   _A_add_entry (char *name, _A_uniqueid& id,
             _A_ENTRYDEF& def, _A_ENTRYTYPE type, _A_D_LEVEL scope)
      // Add an entry with the given "name", "id", and
      // "definition" (created previously) of "type" to the
      // dictionary at the level "scope".  After completion,
      // "def" will be empty, ie. it becomes of _A_NDEF.
      // An add operation can fail if it violates dictionary
      // consistency rules. If it is successful, _A_TRUE is
      // returned, otherwise _A_FALSE is returned.


_A_BOOLEAN  _A_delete_entry (char *name, _A_uniqueid& id,
                      _A_ENTRYTYPE type, _A_D_LEVEL& scope)
      // Delete an entry with the given "name", of "type" in
      // the dictionary at the level "scope".
      // If "name" is NULL or empty string, "id" is used for search.
      // A delete operation can fail if it violates dictionary
      // consistency rules. If it is successful, _A_TRUE is
      // returned, otherwise _A_FALSE is returned.
      // After completion, "scope" has the value of the actual
      // level in which the entry is deleted.


_A_BOOLEAN  _A_lookup (char *name, _A_uniqueid& id, _A_ENTRYTYPE type,
                  _A_D_LEVEL& scope, _A_ENTRYDEF& def)
      //  Given the "name", OR a unique "id" (while "name" is
      //  NULL or an empty string"), and its "type", this function
      //  finds the corresponding entry beginning its search in
```

```
   //  the "scope" sub_dictionary.
       //
       //  This function returns the corresponding "id" or "name",
       //  entry "definition" and "scope" at which it was found.
       //  (i.e. if the "name" is not found in the "scope"
       //  sub-dictionary, it searches in all sub-dictionaries up to
       //  and including the SYSTEM sub-dictionary.)
       //
       //  If "name" is NULL or an empty string, "id" is used as the
       //  search key.
       //  "def" can be anything before look-up, and after look-up
       //  succeeds, the old definition is overwritten by the new one.
       //  _A_TRUE is returned for success, and _A_FALSE for failure.
       //
       //  Lookup will fail if it is a predefined class, eg. CLASS,
       //  MAP, since there is no definition for them.


_A_BOOLEAN  _A_check (char *name, _A_uniqueid& id, _A_ENTRYTYPE type,
                    _A_D_LEVEL& scope)
       // The difference between this function and _A_lookup() is
       // that the definition is not passed back. For pre-defined
       // classes, not like _A_lookup(), _A_TRUE will be returned.


_A_CLSTYPE  _A_classtype (char* name, _A_uniqueid& id,
                         _A_D_LEVEL& scope)
       // This function can find the class type of an instance.
       // One of the values _A_CLS, _A_ATTR, _A_MAP ,_A_SETCLS
       // and _A_NCTYPE is returned. _A_NCTYPE always indicates
       // something is not found in the dictionary and usually is
       // not supposed to be returned.


_A_BOOLEAN  _A_rescope (char *name, _A_uniqueid& id, _A_ENTRYTYPE type,
                      _A_D_LEVEL oldscope, _A_D_LEVEL newscope)
       // rescope the dictionary entry of "type" with "name", from
       // its "oldscope" to "newscope". "oldscope" is only the
       // starting scope for searching.
       // If "name" is NULL or empty, "id" is used for search.
       // Readily, a rescope operation can fail if it violates
       // dictionary consistency rules. If so, _A_FALSE is returned,
       // otherwise, _A_TRUE is returned.


_A_release_dict ()
       // Release the local copy of the dictionary. All changes to
       // the dictionary are written back if necessary.
       // Note that "attach_dict" and "release_dict" effectively
       // define a transaction. "release_dict" can fail if it
       // violates the concurrency control associated with the
       // dictionary.


_A_uniqueid  _A_const_uid (_A_CLSTYPE type)
       // Get the unique id's for those pre-defined classes.
```

```
// "Type" can be _A_CLS, _A_SETCLS, _A_ATTR, _A_MAP,
   // _A_ATTRSET or _A_MAPSET.
```

4.3.4. Glue Routines

The interface described above is in C++. The ADAMS run-time system and the pre-processor can use this interface because they are (or can be) implemented in C++. However, it can not be used directly by the run-time C code generated by the pre-processor. Therefore the dictionary needs a C version interface for the run-time C code to call. We use "glue routines". These glue routines can be called in C code and they in turn call the C++ routines that actually do the work. Also the two kinds of data structures, i.e. _A_ENTRYDEF and iterators, which are implemented as C++ classes, are defined as C structures in the C version interface. These can be found in a C version header file *dict_cc.h.* Since not everything in the C++ interface is needed at run-time, we only provide the C interface for those routines that we currently think will be needed at run-time.

The following is the summary of the C interface we provide at present: † Note that to save space no descriptions have been provided for these interface procedures. Each corresponds to a matching dictionary method in the preceding section. Here we have used the naming convention that *_A_<method>* denotes a C callable procedure to perform the indicated <*method*>.

```
int  _A_set_deftype (def, type)
_A_ENTRYDEF *def;
_A_DEFTYPE  type;

int  _A_add_validm (def, mname)
_A_ENTRYDEF *def;
char        *mname;

int  _A_add_fetchm (def, mname)
_A_ENTRYDEF *def;
char        *mname;

int  _A_add_storem (def, mname)
_A_ENTRYDEF *def;
char        *mname;

int  _A_add_otherm (def, mname)
_A_ENTRYDEF *def;
char        *mname;

int  _A_add_udf (def, udfstr)
_A_ENTRYDEF *def;
char        *udfstr;

int  _A_add_ukn (def, uknstr)
_A_ENTRYDEF *def;
char        *uknstr;

int  _A_add_superclass (def, name_str, level)
_A_ENTRYDEF *def;
```

---

† All type definitions are in the C interface header file *dict_cc.h.*

```
char            *name_str;
    _A_D_LEVEL  level;

    int  _A_add_elementclass (def, name_str, level)
    _A_ENTRYDEF *def;
    char        *name_str;
    _A_D_LEVEL  level;

    int  _A_add_image (def, name_str, level)
    _A_ENTRYDEF *def;
    char        *name_str;
    _A_D_LEVEL  level;

    int  _A_add_attrset (def, setname, id, lv, syn)
    _A_ENTRYDEF *def;
    char        *setname;
    char        *id;
    _A_D_LEVEL  lv;
    char        *syn;

    int  _A_add_mapset (def, setname, id lv, syn)
    _A_ENTRYDEF *def;
    char        *setname;
    char        *id;
    _A_D_LEVEL  lv;
    char        *syn;

    int  _A_add_booleanm (def, mname)
    _A_ENTRYDEF *def;
    char        *mname;

    int  _A_set_codomain (def, name_str, level)
    _A_ENTRYDEF *def;
    char        *name_str;
    _A_D_LEVEL  level;

    int  _A_add_subvalue (def, mname)
    _A_ENTRYDEF *def;
    char        *mname;

    int  _A_add_instclass (def, name_str, level)
    _A_ENTRYDEF *def;
    char        *name_str;
    _A_D_LEVEL  level;

    int  _A_attach_dict (tid, uid, file_name, pre_processor)
    unsigned tid, uid;
    char     *file_name;
    int      pre_processor;

    int  _A_c_add_entry (name, id, def, type, scope)
    char         *name;
    char         *id;
    _A_ENTRYDEF  *def;
    _A_ENTRYTYPE type;
    _A_D_LEVEL   scope;
```

```
int  _A_c_delete_entry (name, id, type, scope)
char         *name;
char         *id;
_A_ENTRYTYPE type;
_A_D_LEVEL   *scope;

int  _A_c_lookup (name, id, type, scope, def)
char         *name;
char         *id;
_A_ENTRYTYPE type;
_A_D_LEVEL   *scope;
_A_ENTRYDEF  *def;

int  _A_c_check (name, id, type, scope)
char         *name;
char         *id;
_A_ENTRYTYPE type;
_A_D_LEVEL   *scope;

int  _A_c_rescope (name, id, type, oldscope, newscope)
char          *name;
char          *id;
_A_ENTRYTYPE  type;
_A_D_LEVEL    oldscope;
_A_D_LEVEL    newscope;

int  _A_release_dict ()

int  _A_c_const_uid (id, classtype)
char       *id;
_A_CLSTYPE classtype;
```

4.3.5. An Example

Below is an ADAMS program that exercises many of the dictionary features. In it we follow the convention of capitalizing class names and using lower case for instance names. However nothing in the ADAMS language, preprocessor or dictionary is case dependent.

```
#include <stdio.h> main ()
{
<< open_adams 3   >>

<< X isa MAP with image Y, scope is task  >>
<< x_inst instantiates_a X    >>

<< A isa Class , having {attr1_inst}      >>
<< a_inst instantiates_a A    >>

<< C isa A and B, having x = {attr1_inst} , scope is system >>

<< D isa attribute, with image FOO having {y}   >>
<< d instantiates_a D         >>

<< S isa set of FOO elements, scope is user     >>
<< s_inst instantiates_a S    >>
```

```
<< Z isa CODOMAIN, consisting of #a-zA-Z# , scope is TASK >>

    << close_adams 3  >>
    }
```

The following is the code generated by the preprocessor for run-time execution:

```
    _A_attach_dict(task_id, user_id, "tests", 0);
    {
    char        _A_id[9];
    _A_ENTRYDEF _A_def;
    _A_def.type = _A_NDEF;  /* For each _A_ENTRYDEF variable, this   */
                            /* must be done before they can be used. */
                            /* This is one of the differences        */
                            /* between C and C++.                    */
                        << X isa MAP with image Y, scope is task >>
                /* Recognize "isa MAP" */
    _A_set_deftype(&_A_def, _A_CLASSDEF);
    _A_add_superclass(&_A_def, "MAP", _A_TASK);

                /* Recognize "with image Y" */
    _A_add_image(&_A_def, "Y", _A_TASK);

    .....       /* get a new uniqueid for this map class */
                /* and put it in "_A_id"             */

    if (_A_add_entry("X", _A_id, &_A_def, _A_CLASS, _A_TASK))
          {           /* Try to add to dictionary    */
                      /* Successful dictionary entry */
          ......
          }
    else
          {           /* Entry could not be made    */
          ......
          }
                        << x_inst instantiates_a X >>
    _A_set_deftype(&_A_def, _A_INSTDEF);
    _A_add_instclass(&_A_def, "X", _A_USER);

    .....       /* get a new uniqueid for this instance   */
                /* and put it in "_A_id"            */

    if (_A_add_entry("x_inst", _A_id, &_A_def, _A_INST, _A_USER))
          {
          ......
          }
    else
          {
          ......
          }
                        << A isa Class , having { attr1_inst } >>
    _A_set_deftype(&_A_def, _A_CLASSDEF);
    _A_add_superclass(&_A_def, "CLASS", _A_USER);

                /* suppose {attr1_inst} is an attribute set that has */
```

```
      /* already been made up by the preprocessor.         */
            /* Its uniqueid is set_id.                              */
_A_add_attrset(&_A_def, NULL, set_id, _A_USER, NULL);

..... /* get a new uniqueid for this class */
      /* and put it in "_A_id"             */

if (_A_add_entry("A", _A_id, &_A_def, _A_CLASS, _A_USER))
      {
      ......
      }
else
      {
      ......
      }
```

<< **a_inst instantiates_a A** >>

```
_A_set_deftype(&_A_def, _A_INSTDEF);
_A_add_instclass(&_A_def, "A", _A_USER);

..... /* get a new uniqueid for this instance */
      /* and put it in "_A_id"                */

if (_A_add_entry("a_inst", _A_id, &_A_def, _A_INST, _A_USER))
      {
      ......
      }
else
      {
      ......
      }
```

<< **C isa A and B, having x = {attr1_inst} , scope is system** >>

```
_A_set_deftype(&_A_def, _A_CLASSDEF);
_A_add_superclass(&_A_def, "A", _A_SYSTEM);
_A_add_superclass(&_A_def, "B", _A_SYSTEM);
_A_add_attrset(&_A_def, NULL, set_id, _A_SYSTEM, NULL);
            /* set_id is the unique id of {attr1_inst} */

.....       /* get a new uniqueid for this class */
            /* and put it in "_A_id"             */
if (_A_add_entry("C", _A_id, &_A_def, _A_CLASS, _A_SYSTEM))
      {
      ......
      }
else
      {
      ......
      }
```

<< **D isa attribute, with image FOO having {y}** >>

```
_A_set_deftype(&_A_def, _A_CLASSDEF);
_A_add_superclass(&_A_def, "ATTRIBUTE", _A_USER);
_A_add_image(&_A_def, "FOO", _A_USER);

                  /* suppose {y} is an attribute set that has */
                  /* already made up by the preprocessor.     */
                  /* Its uniqueid is set_id.                  */
_A_add_attrset(&_A_def, NULL, set_id, _A_USER, NULL);
```

```
..... /* get a new uniqueid for this attribute class */
      /* and put it in "_A_id"                        */

if (_A_add_entry("D", _A_id, &_A_def, _A_CLASS, _A_USER))
      {
      ......
      }
else
      {
      ......
      }
```

<< d instantiates_a D >>

```
_A_set_deftype(&_A_def, _A_INSTDEF);
_A_add_instclass(&_A_def, "D", _A_USER);

..... /* get a new uniqueid for this instance */
      /* and put it in "_A_id"                */

if (_A_add_entry("d", _A_id, &_A_def, _A_INST, _A_USER))
      {
      ......
      }
else
      {
      ......
      }
```

<< S isa set of FOO elements, scope is user >>

```
_A_set_deftype(&_A_def, _A_CLASSDEF);
_A_add_superclass(&_A_def, "SET", _A_USER);
_A_add_elementclass(&_A_def, "FOO", _A_USER);

..... /* get a new uniqueid for this set class */
      /* and put it in "_A_id"                  */

if (_A_add_entry("S", _A_id, &_A_def, _A_CLASS, _A_USER))
      {
      ......
      }
else
      {
      ......
      }
```

<< s_inst instantiates_a S >>

```
_A_set_deftype(&_A_def, _A_INSTDEF);
_A_add_instclass(&_A_def, "S", _A_USER);

..... /* get a new uniqueid for this instance */
      /* and put it in "_A_id"                */

if (_A_add_entry("s_inst", _A_id, &_A_def, _A_INST, _A_USER))
      {
      ......
      }
else
      {
```

```
......
      }
```

<< Z isa CODOMAIN, consisting of #a-zA-Z# , scope is TASK >>

```
_A_set_deftype(&_A_def, _A_CODEF);
_A_add_validm (&_A_def, "....");          /* #[a-zA-Z0-9]# */

..... /* get a new uniqueid for this codomain */
      /* and put it in "_A_id"                 */

if (_A_add_entry("Z", _A_id, &_A_def, _A_CO, _A_TASK))
      {
      ......
      }
else
      {
      ......
      }

set_deftype(&_A_def, _A_NDEF);      /* this must be done before   */
                                    /* leaving the scope of "def" */
}
```

4.4. Dictionary Implementation

The current ADAMS dictionary is implemented upon the unique ID module of the ADAMS system, and the only part of the unique ID module interface the dictionary uses is the unique ID storage routines. That is because everything in ADAMS is stored through the unique ID storage mechanism.

4.4.1. Representation of the Dictionary in Memory

Figure 4.2 shows the logical structure of a sub-dictionary. This is the dictionary representation *in memory*. Every sub-dictionary has a level flag indicating its level, ie. SYSTEM, TASK, USER or LOCAL, a header and some pointers to some entry lists. The four types of entries are put in four separate lists. These lists contain only the name parts of entries as well as minimal bookkeeping information. Therefore they are called *name-lists.* The elements in the name-lists are blocks (or "chunks") called *list-chunks*, each of which contains a header, an entry array, and one corresponding pointer arrays. Each element in the entry array contains the name part of an entry and bookkeeping information about that entry. The pointer array is a definition pointer array. Each element in this array is a pointer to the definition part. Of course not all the definitions need be in memory for any simple program. Therefore some of the pointers can be null. Only when needed, will a definition part be fetched into memory and the corresponding pointer is set to it.

4.4.2. Persistent Storage of Name Parts

Now we know that the name parts of all entries are put into some chunks and these chunks are linked together to form the basic structure of a sub-dictionary. Since the dictionary is persistent, we have to store sub-dictionaries into disk. To do this, we first set up a Unix directory under which the dictionary is to be stored. The sub-directories in this directory have the structure shown in Figure 4.3.

We can see that each persistent sub-dictionary has five files: *hd* (header), *co* (codomain), *cl* (class), *sp* (subscript pool) and *in* (instance). The sub-dictionary header is saved in the first file. The other four files contain the list-chunks of the four types. Since we assign every chunk a

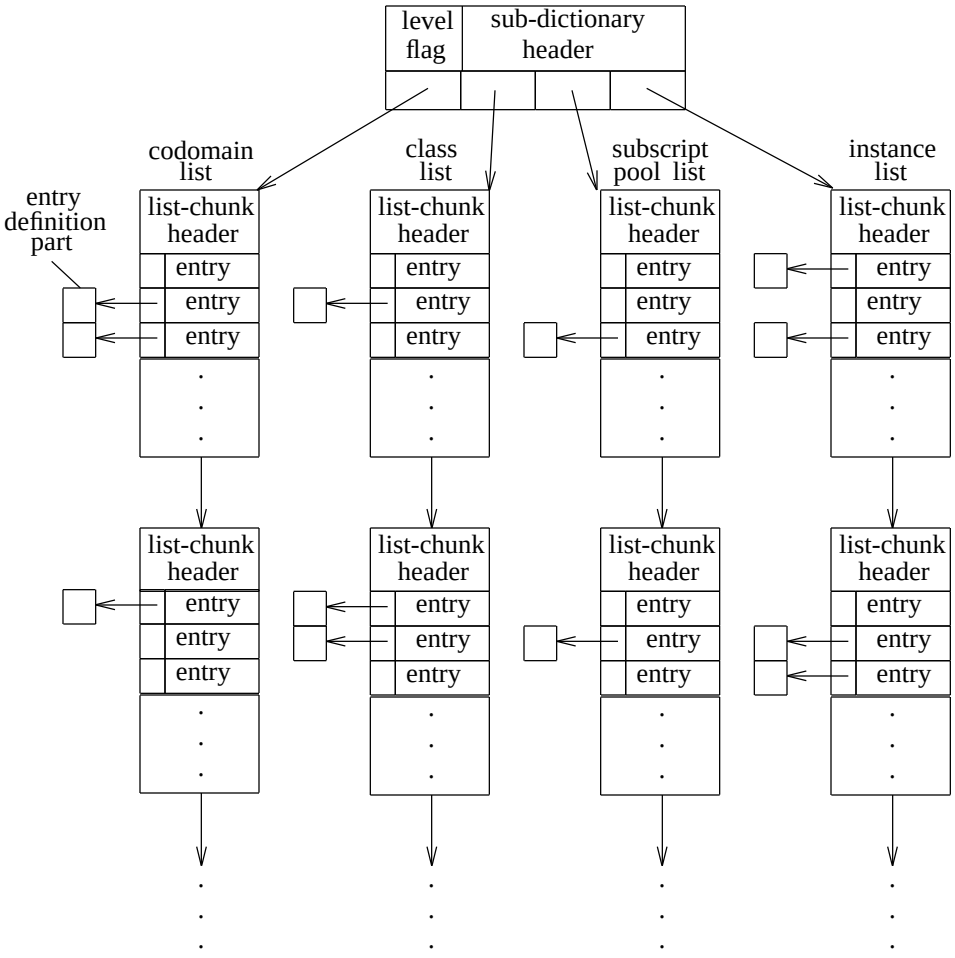

Figure 4.2 Sub-dictionary Structure

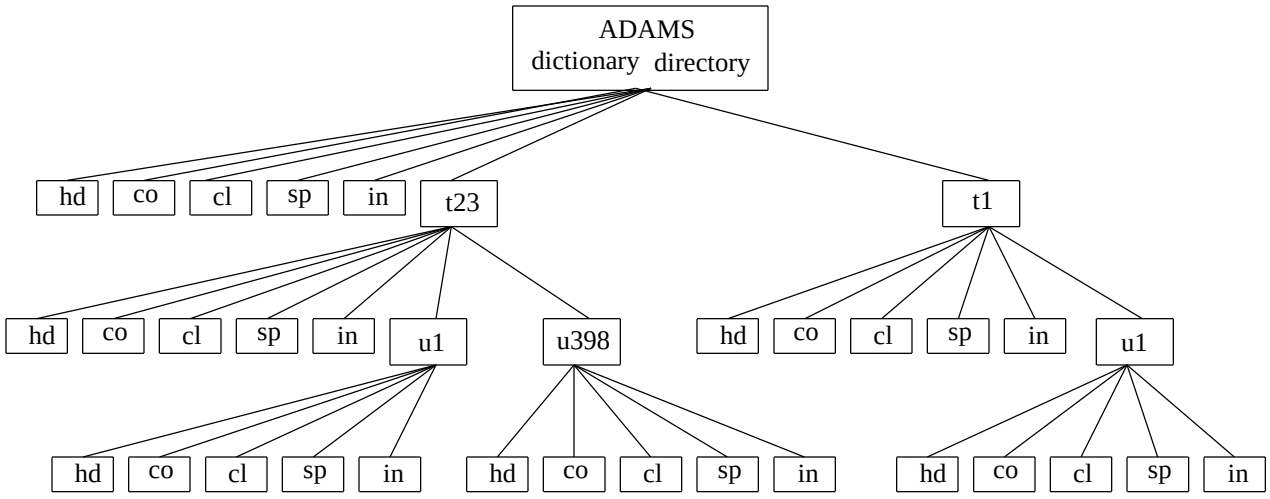

Figure 4.3 Structure of ADAMS Dictionary Directory

number when we create them, all the chunks are stored in increasing order in the files they belong to. In the *hd* file of the SYSTEM sub-dictionary, besides the header, the unique ID's of the six pre-defined classes are also stored at the end of the file. They are obtained when the SYSTEM sub-dictionary is created and are never changed.

4.4.3. Representation and Storage of Definition Parts

The four types of entry definitions are defined as four C++ classes. When a new definition is to be created, an _A_ENTRYDEF object is first filled out. After this is finished, the definition data is added into the dictionary.

We know that in a class definition there may be a list of super class unique ID's and lists of associated sets. Also in an instance definition there may be a list of super class unique ID's. When an entry is being defined, ie. an _A_ENTRYDEF object is being filled out, the number of elements in these lists are unknown until the process is completed. So we have to put them in some temporary lists. But once the entry is defined, these lists will never be changed. Therefore when the entry is added to the dictionary, these fields are copied from the temporary lists into some dynamic arrays which are of the exact sizes to hold the the data. This saves space since no pointer is needed to link them. This is done by the two member functions of _A_ENTRYDEF, i.e. *_A_ENTRYDEF::move_classes()* and *_A_ENTRYDEF::move_assocsets().*

Unlike the list-chunks, the definition part of each entry is not saved in dictionary files. Since every entry has a unique ID, and in our implementation unique ID's are objects and have disk space allocated for them, we use the ADAMS unique ID storage mechanism to store the definition part of entries. (Eventually the list-chunks can also have unique ID's and be stored in this way. Therefore in a later implementation the dictionary files *co, cl, sp* and *in* may disappear. We will talk about this more later in our discussion of further work.)

Before a definition is to be saved in the disk, it must first be put into an internal buffer in some format and then all the bytes are flushed into disk by invoking the unique ID store function. When a definition is fetched into memory, we first get it into an internal buffer by calling the unique ID load function. Then, since we know the format in which we stored it, we can convert these bytes of data into a run-time structure. Doing so, we need to know the number of bytes

needed to store a definition. This number is kept in the _A_ENTRYDEF object when it is being filled out. Then it is written into the name part of the entry in list-chunk. The formats we use for the four types of definitions are as follows. Compare these to the corresponding logical lists in Section 4.2.

Codomain definition:

validating method	*(null character terminated string)*
fetch method	*(null character terminated string)*
store method	*(null character terminated string)*
number of other methods	*(1 byte)*
other methods	*(null character terminated strings)*
undefined value	*(null character terminated string)*
unknown value	*(null character terminated string)*

Class definition:

number of super classes	*(1 byte)*
super class unique ID's	
attribute or map image unique ID	
set element class unique ID	
number of associated attribute sets	
associated attribute set unique ID	
associated attribute set synonym	*(null character terminated string)*
associated attribute set unique ID	
associated attribute set synonym	*(null character terminated string)*
...	
number of associated map sets	
associated map set unique ID	
associated map set synonym	*(null character terminated string)*
associated map set unique ID	
associated map set synonym	*(null character terminated string)*
...	
boolean method	*(null character terminated string)*

Subscript Pool:

codomain unique ID	
number of values	*(1 byte)*
pool values	*(null character terminated strings)*

Instance:

number of super classes	*(1 byte)*
super class unique ID's	

4.4.4. Implementation

Before the dictionary can be used, it should be "attached" to first by calling the routine *_A_attach_dict()*. What is actually done is that first some global variables are set. One of them is the flag indicating if the pre-processor or a user program is running. Then several run-time sub-dictionary structures are created and the sub-dictionaries of all the three persistent levels are loaded into memory. In the current implementation, all the list-chunks are loaded. If it is the pre-processor that is running, a set of shadow sub-dictionaries are also created. So when new

entries are added into the dictionary, they will actually be added into the shadow sub-dictionaries to keep the dictionary unchanged at pre-processing time.

When a program is done with the dictionary, the dictionary should be released by calling *_A_release_dict()*. If the pre-processor is running, all to be done is just releasing the memory space allocated for the dictionary. Otherwise, all changes to the dictionary should be written back to the permanent storage first.

When an entry is added to the dictionary (only _A_ENTRYDEF objects can be used to add an entry), some consistency conditions should be examined first. In the current implementation, we only check to make sure that there is not another entry of the same type with the same name at the same or higher level. If an empty slot in the sub-dictionary is found, the new entry is added there; otherwise, a new list-chunk is created and the new entry is inserted there.

An entry can be deleted only if it is not being referenced by any other entries. In the current implementation, every entry has a reference count which keeps the number of entries that are referencing it. We can delete an entry as long as its reference count is zero. While deleting an entry, we should also decrease the reference counts of all the entries to which this entry has a reference.

For rescoping an entry, only upward rescoping is allowed in current implementation and an entry can be rescoped to a higher level only if there is not another entry of the same type with the same name in a higher level than the old scope and all the entries referenced by this entry are still visible, ie. inclusively above the new scope.

4.5. Programs for Creating and Removing a Dictionary

When the system is first set up, we need commands to create an empty SYSTEM sub-dictionary. Later as new tasks and new users come in, we also need to create empty sub-subdictionaries for them. When tasks are canceled or users go away, their sub-dictionaries should be removed. To do all this, we have two command programs to create and remove sub-dictionaries.

The command program to create a sub-dictionary is *createdict.* The information needed to create a sub-dictionary includes the dictionary directory, the level of the sub-dictionary to be created and the necessary task ID number and user ID number. The dictionary directory can be specified by a command line option. If no option is given, the dictionary directory is obtained from the environment variable DICTDIR in the user's *.profile* file. The other information is given as command line arguments. In addition, the necessary dictionary sub-directories must first exist before the sub-dictionary can be created. For example, suppose our dictionary directory is /at0/adams/TEST. To create up the SYSTEM sub-dictionary, the directory /at0/adams/TEST must first exist. Then we give the command:

```
$ createdict s                # create system dictionary files
                              # DICTDIR=/at0/adams/TEST
```

or

```
$ createdict -d/at0/adams/TEST s
```

Now if we want to create a sub-dictionary for a new task with ID number 123, the directory */at0/adams/TEST/t123* is first created using the command:

```
$ createdict t 123
```

or

```
$ createdict -d/at0/adams/TEST t 123
```

To set up the sub-dictionary for a new user with user ID 567 in task 123, we must also create the directory */at0/adams/TEST/t123/u567*. The command to create this user sub-dictionary is:

```
$ createdict u 123 567
```
or
```
$ createdict -d/at0/adams/TEST u 123 567
```

To remove the dictionary, run the program *removedict.* The command line arguments are exactly the same as for *createdict,* e.g.

```
$ remove -d/at0/adams/TEST u 123 567
```

4.5.1. Implementation of *createdict* and *removedict*

Before a sub-dictionary is created, *createdict* first checks to make sure that the specified sub-dictionary already exists. If so, nothing is done; otherwise, *SUBDICT::set_storage()* is called to create the sub-dictionary files, ie. *hd, co, cl, sp* and *in.* The header of the empty sub-dictionary is written into *hd.* If it is the SYSTEM sub-dictionary, the unique ID routine is called to get six unique ID's for the six pre-defined classes and they are also written into *hd.* These unique ID's will never be assigned to any other objects and they always represent the pre-defined classes.

To remove a sub-dictionary, the name-lists are first loaded into memory so that the unique ID's of each entry could be used to release the permanent storage for definitions, because the definition of each entry is stored by unique ID store routine. After that, all the sub-dictionary files are removed.

4.6. Interactive Dictionary Test Program

An interactive dictionary test program was created to validate various dictionary functions. It currently has the capability to insert, delete, look up, and rescope dictionary entries, print out sub-dictionaries, and give the upper most super class of an instance, i.e. one of the six pre-defined classes. By modifying the code future extensions can be similarly tested without affecting the existing ADAMS system. This program is in the file *test1*. To run it, just type the command

```
$ test1 -d/at0/adams/TEST -t111 -u222
```

All the command line arguments can be omitted. The default tid is 123 and uid 567. The default dictionary directory is read from the environment variable DICTDIR in the user's *.profile* file. No order is enforced for these arguments. Here is an example of running the program (user responses are denoted in *italics*):

```
$ test1
program name > my_program
pre_processor(y/n)> n
Enter function: Insert Rescope Delete Print Lookup Classtype Quit> i
Enter level System, Task, User, Local, quit > t
Enter type Domain, Class, Subscript, Instance > d
Enter name > our_codomain
Enter validating method name > its_valm
Enter fetch method name > its_fetm
Enter store method name > its_stom
Enter other method name > its_othm
Enter other method name > another_othm
Enter other method name >
Enter udf > its_udf
Enter ukn > its_ukn
Enter name >
Enter function: Insert Rescope Delete Print Lookup Classtype Quit> q
$
```

The program first asks for your program name. It is not actually used in the current implementation, so you can enter anything you want. Then it asks if it should run as if the preprocessor or the user program were having access to the dictionary. Here we answered *n*, which means all changes will be written to the permanent dictionary just as if the user program were running. Then a top level function list is given. To select one of them, only the first character is needed. We selected *insert.* Then the level at which the new name is to be inserted is asked. We gave *task.* Then what kind of name? Ours is a codomain. The codomain name is *our_codomain.* Then all its method names are entered. The program will keep on asking for next "other method" until an empty string is entered. After this codomain is defined, it is inserted into the dictionary and the program goes on to ask for another codomain until an empty line is entered. Then the codomain insertion is finished and the top level function list is displayed again.

Now we want to look up this codomain:

```
$ test1
program name > another_prog
pre_processor(y/n)> y
Enter function: Insert Rescope Delete Print Lookup Classtype Quit> l
Enter type Domain, Class, Subscript, Instance) > d
Enter lookup name > our_codomain
Enter level of search System, Task, User, Local, quit > l
lookup 0-type, starting level-0, entry_str = our_codomain
            FOUND at  TASK level
validating method:  its_valm
fetch method:  its_fetm
store method:  its_stom
other method(s):  another_othm
                  its_othm

udf: its_udf
ukn: its_ukn
Enter function: Insert Rescope Delete Print Lookup Classtype Quit> q
$
```

Note that in the above example, we responded with search level *local* although *our_codomain* is at the TASK level. This means the search will begin from LOCAL and go upwards.

The process to run other functions is quite similar and the program is quite self-explanatory.

4.7. Suggestions for Further Work

(1) In the current implementation, all the list-chunks are in the memory. This can be improved if we have two lookup tables for each sub-dictionary. Every list-chunk can be given a unique ID. One of the tables converts the name of an entry to the list-chunk ID in which it resides. The other table converts the unique ID of an entry to the corresponding list-chunk ID. By these two tables, given either a name or a unique ID, we can find the list-chunk ID in which this entry resides. These two tables can be implemented as O-trees. Then not all the list-chunks have to be in memory. Only when a name is looked up and it is not found in memory, will its list-chunk be loaded into memory. When a new entry is inserted into dictionary, it will be inserted into those list-chunks that are already in memory if they have an empty slot; otherwise, a new list-chunk is created and the name is inserted there.

(2) The subscripted name feature is not implemented yet. If a name has a subscript, it will be denoted by, besides its unique ID, a number which is calculated from its subscript. So

given a subscripted name, the task for the dictionary is to find its unique ID and calculate this number from the subscripts.

(3) The parameterized class has to be implemented.

(4) The rescoping mechanism has not been implemented completely. The key issue is to keep reference counts. An entry in SYSTEM or TASK level need multiple reference counts, one for each task or user who uses this entry. To represent and store these reference counts efficiently is what we need to solve. This is essentially a searching problem. Given a user ID, or a task ID, or both, we should be able to find the corresponding reference count for that entry. Different approaches can be examined, such as balanced binary tree etc.

(5) A robust error handler has to be developed. Currently our error routine just prints out the error message to the standard output. A robust handler should meet the needs of the pre-processor. This could be achieved while the pre-processor is developed further.

5. Low-Level Representations

5.1. Uniqueid Identifier Representation

Uniqueids are central to the implementation of ADAMS. From the end-user's viewpoint, they identify every ADAMS element; from the implementer's viewpoint, they identify every individual piece of low-level storage. Because of this pervasiveness, uniqueids form the natural medium for the interfaces between the user's codes and the run-time system, and between the run-time system and the Transaction Manager and low-level storage.

5.1.1. User Interface to Uniqueids

The end-user sees a very simplified and restricted view of uniqueids. The definition that he uses is simply a null-terminated character string of eight printable characters:

```
typedef char _A_uid_string[9]
```

The only operations available to the end-user are the following:

```
_A_uniqueid_getuid  (_A_uid_string uid)
      //  Sets 'uid' to a new uniqueid value.

_A_uniqueid_instant  (_A_uid_string uid, _A_uid_string dict_entry)
      //  Instantiates 'uid', initializing its link count and
      //  retaining 'dict_entry' as a pointer back to the
      //  CLASS of 'uid.'
```

These functions are not called directly by the end user, but are instead inserted by the preprocessor. The preprocessor also uses uniqueids as handles to the attributes, maps, and sets that are maintained by the Index Manager.

5.1.2. Run-time Uniqueids

Virtually all actions taken by the run-time system involve uniqueids. The run-time representation of uniqueids is defined by the C++ class *_A_uid*. The only physical structure associated with an object of this class is an eight-character array; underlying the *_A_uid*s, however, there is a uniqueid manager, the *_A_uidmgr* class, which interfaces with the Transaction Manager and low-level storage.

The run-time system views uniqueids in two different ways: as identifiers of ADAMS elements, and as identifiers of storage locations. In either case, there are some basic operations on uniqueids that are included as member functions of the class *_A_uid*:

```
_A_uniqueid ()
      //  This constructor creates a uniqueid and sets it equal to
      //  the constant _A_NULLUID, an empty string.

_A_uniqueid (const &_A_uniqueid)
      //  This constructor creates a new uniqueid and sets it equal
      //  to its argument uniqueid.

_A_uniqueid (char*)
      //  Given a character string, this constructor creates a new
      //  uniqueid and initializes it to the argument string.
      //  This cis used mainly for debugging programs and is not
      //  for use by the actual ADAMS run-time system.

void disp ()
      //  This function is used by debug statements to display the
```

```
  //  contents of a uniqueid.

void uidcopy ()
      //  This function copies a null-terminated uniqueid to a string.
      //  It is used by some of the glue routines.
```

Comparison Operators: Uniqueids can be lexicographically compared to each other using the overloaded comparison operators ==, != and >. Comparisons return *_A_TRUE* or *_A_FALSE*.

5.1.2.1. Element Identifiers

For every ADAMS element, the run-time system must maintain an instance record, consisting of at least a link counter and an indicator of the element's CLASS. In order to achieve this, the system maintains the *aidmgr*, a large index that maps _A_uids to the location of the instance record. _A_uid member functions associated solely with this view of uniqueids are:

```
void instantiate (_A_uniqueid& dict_entry)
      //  Instantiates an ADAMS element by creating storage for
      //  its link count and the uniqueid of its dictionary class.
      //  Inserts an entry in the aidmgr mapping the instantiated
      //  uniqueid to the new storage location.

void inc_link_count ()
      //  Accesses the aidmgr and increments link count of the uniqueid.

void dec_link_count ()
      //  Accesses the aidmgr and decrements link count of the uniqueid.

_A_BOOLEAN element_exists ()
      //  Accesses the aidmgr and returns _A_TRUE if an instance
      //  record for the uniqueid is found;
      //  otherwise, it returns _A_FALSE.
```

5.1.2.2. Storage Location Identifiers

In implementing ADAMS, uniqueids were used to refer to numerous pieces of storage that are invisible to the end-user, such as (element uniqueid, codomain value) pairs, index blocks, and dictionary entries. In many cases, the most convenient uniqueid to use for this identification happened to also be an ADAMS element identifier. The best example of this is a set, which is an ADAMS element that must have some extra storage—an index—associated with it. In order to keep the set index separate from the instance information a separate storage manager was created.

The storage manager, or *sidmgr*, maps uniqueids to storage locations associated with ADAMS elements' data. These locations store the (element uniqueid, codomain value) pairs pointed to by indexes used to implement maps, attributes, and sets, and pieces of the indexes themselves.

In the case of maps and attributes, the dictionary also needs to store some information concerning the uniqueid. In order to allow the dictionary to also store data under the uniqueid, and to prevent any clashes, the dictionary has its own storage manager, the *didmgr*. It is identical to the *sidmgr*, except that the locations it points to are used to store information for the dictionary alone. _A_uid member functions related to storage are

```
_A_BOOLEAN storage_exists ()
      //  Accesses the sidmgr and returns _A_TRUE if a storage
      //  location is found for the uniqueid;
      //  otherwise, it returns _A_FALSE.
```

```
int fetch (_A_STORE_TYPE mgr, char* buffer, int buf_length)
      //  Accesses either the didmgr or sidmgr, depending upon the
      //  value of 'mgr,' and fetches the data stored at the
      //  uniqueid, writing it into 'buffer'.
      //  Returns the actual number of bytes fetched.

int store (_A_STORE_TYPE mgr, char* buffer, int buf_length)
      //  Updates the didmgr or sidmgr, as indicated by the
      //  value of 'mgr,' and stores the contents of 'buffer'
      //  under the uniqueid.
      //  Returns the number of bytes stored.

void unstore (_A_STORE_TYPE mgr)
      //  Frees memory associated with an entry in either the
      //  didmgr or sidmgr, as indicated.
      //  Also removes the uniqueid from the table.
```

This section describes our current implementation of the *uid* concept. As we have discovered, we can eliminate any direct representatin of them and treat *uid*'s as virtual tokens with their reference counters and class pointers functionally treated as *attributes* and *maps* respectively.

5.2. Storage

One goal of ADAMS has been to eliminate the traditional file/record structures as the paradigm of persistent data storage. Programmers have historically accessed persistent data through filenames and relative locations such as a byte offsets or record numbers. An unfortunate consequence of this approach is that migration of data either greatly sacrifices efficiency or is altogether unworkable; processes that need to access the data cannot keep track of its location. In addition, programmers must handle the low-level file I/O necessary to operate on the object, concurrent access to data cannot be supported without great care and operating system support. These weaknesses can largely be overcome by the client-server model, where by a *client* we mean any process which uses *bags* to store and access persistent data. These client processes will invariably be ADAMS preprocessing or run-time procedures since bags are *invisible* to applications programs.

Client processes refer to persistent data through some form of storage-independent handles, while a server is responsible for mapping handles to actual disk locations. Because an item's handle is invariant over its lifetime, the server can move data arbitrarily so long as its mapping rules are revised appropriately.

The ADAMS Storage Management System is based on this model. Clients are provided with an interface that allows only high-level operations which we describe in this section, while the server performs all low-level I/O in response to client requests. (Details of server implementation can be found in [Jan89].) This approach provides client processes with a small set of functions that support an abstract data type called a *bag*, a persistent collection of variable-length or fixed-size objects. Individual bags are not bound to any storage device. While this implementation of the Storage Manager does not concern itself with migration, we will that show the approach used is easily extensible to provide this capability.

Client requests are received by a server, which utilizes a group of I/O-dedicated processes to perform the actual I/O, as shown in Figure 5.1. The server operates as a daemon that can multiplex many clients.

Under this system, a client uniquely identifies a data item by a (*bag_number, item_number*) pair. The latter is simply a secondary tag that uniquely identifies this particular element in the bag, but not its location on a storage device. A client can store an item by creating a bag and then

Schematic of bag storage management.
Figure 5.1

inserting the item into it. Those operations return a bag number and an item number respectively; the client can later access the item by these values. Other functions allow retrieval, modification and deletion of items, and deletion of bags.

A *client* is any process which accesses data represented in bags. Clients view bag operations as function calls. The function calls mask the underlying system calls required to implement the client-server interprocess communication (IPC). We use long integer bag and item identifiers, which we type as *BAGNO* and *ITEMNO*.

There are six functions provided to clients to operate on bags and items. The declarations and semantics of the functions are listed below. All functions return negative values on error. Error conditions are listed in Appendix A of [Jan89].

```
BAGNO create_bag (long length)
      /*
      **  Creates a new bag and returns its bag number.
      **  If length is positive, the bag may contain
      **  only items of this specified number of bytes.
      **  A non-positive value specifies that the bag
```

```
   **  may contain variable-length items.
       */

ITEMNO insert_item (BAGNO b, char *s, long length)
       /*
       **  Inserts the sequence of length bytes beginning at address s
       **  into bag b, and returns the item number assigned
       **  to the stored bytes. If the bag was created for fixed-length
       **  items, the length must match the value originally
       **  passed to create_bag().
       */

long retrieve_item (BAGNO b, ITEMNO i, char *s, long length)
       /*
       **  Retrieves the bytes stored as item i in bag b.
       **  Copies at most length bytes to the buffer beginning at
       **  address s. Returns the actual size of the item in bytes.
       */

long modify_item (BAGNO b, ITEMNO i, char *s, long length)
       /*
       **  Replaces item i in bag b with the sequence of
       **  length bytes beginning at address s. If the bag was created
       **  for fixed-length items, length must match the value
       **  originally passed to create_bag().
       */

long delete_item (BAGNO b, ITEMNO i)
       /*
       **  Deletes item i from bag b.
       **  The item number can later be reassigned by the server
       **  when another item is inserted into bag b
       **  via insert_item().
       */

long delete_bag (BAGNO b)
       /*
       **  Deletes bag b.
       **  There is no requirement that the bag be empty.
       **  The bag number can later be reassigned by the server
       **  when another bag created via create_bag().
       */
```

Two functions are provided to allow the client to arbitrarily open and close a connection with the server. This ability provides significant flexibility to programmers concerned with robustness. For example, if the server crashes and is then rebooted, clients that were not connected over this period will be unaffected.

As with the bag functions, these connection functions return negative values if they fail.

```
int open_connection ()
       /*
       **  Opens a communication channel between client and the server.
       */

int close_connection ()
       /*
       **  Closes the communication channel between client and the server.
```

```
   */
```

Another function allows the client to determine if the connection is open or closed. While the state can also be determined by the return values of the *open_connection()* and *close_connection()* functions, this function provides a convenient mechanism for code with multiple entry points, such as signal handlers, to test connection status:

```
int connected ()
      /*
      **  Returns a nonzero if the communication channel between the
      **  client and server is open, and zero otherwise.
      */
```

When any of the functions fail, the external variable *errno* is set to the error number. A mechanism exists so the user can access a string that describes the error:

```
char *errstr ()
      /*
      **  Returns a pointer to a string describing a client error.
      */
```

Most UNIX system calls can alter the value of *errno*, so this function should be invoked immediately after a Storage Manager function fails, as in the following code fragment:

```
if (open_connection() < 0)
      {
      fprintf (stderr, "Open_connection failed: %s\n", errstr());
      exit (1);
      }
```

A process may have no more than one connection to the server open at a given time. A call to open the connection when it is already open, or to close the connection when it isn't open will fail. A process with the connection open can exit without calling *close_connection()* — the kernel will close the client's side of the connection when cleaning up the process' open file table. Since file descriptors are shared over a *fork()* system call, a client program that forks while connected should close the connection in either the parent or child. Since file descriptors are inherited over *exec()* system calls, a client should always close the connection before an *exec()*.

Client processes have full control to decide when bags should be created and what is to be inserted into them. The intention is that each bag should contain related items likely to be accessed sequentially, and different bags should be used for collections of data that are likely to be accessed in parallel. Later versions of the ADAMS Storage Manager will be optimized for those circumstances. Clearly, the determination of relatedness, while varying for different client applications, is better decided at a higher level.

Client processes also have full control over when the communication channel to the server is opened. A client process that infrequently needs access to data managed by the server should open the connection only when necessary. The maximum number of connections that the server can have open is finite, though large enough to handle many clients simultaneously.

This storage management module was written in C. The C++ user simply links his object file(s) with the object file containing the connection and bag-related functions. A file containing the type definitions for BAGNO and ITEMNO and the declarations of the functions and their arguments must be included into any client source files that call the functions to satisfy the strict type-checking requirements of *cfront*, the C++ syntax and type-checker. The C user can link with the same object file, but a C-style *include* file must be used instead of the C++-style file.

6. Codomains

In this implementation, all codomains are represented as ASCII strings that are coerced into appropriate host language types by methods attached to the fetch and store operators. More general codomains will be implemented in the next version.

7. Indexes — Functions and Aggregate Structures

Many relationships between associated elements of a database must be maintained by the database manager. In a general mathematical sense, all such relationships can be regarded as a collection of ordered pairs ($element_{A,}$ $element_{B)}$, indicating that $element_A$ (of type *A*) is related to $element_k$ (of type *B*) by the relation R(A,B) defined on the classes (types) *A* and *B*. Such a general relationship can be implemented as a simple table of the form

$$
\begin{array}{l}
(elem_{A_1}, elem_{B_1}) \\
(elem_{A_1}, elem_{B_j}) \\
\quad . \qquad . \\
\quad . \qquad . \\
(elem_{A_i}, elem_{B_j}) \\
(elem_{A_i}, elem_{B_k}) \\
\quad . \qquad . \\
\quad . \qquad . \\
(elem_{A_m}, elem_{B_n})
\end{array}
$$

Such a structure can be thought of as an "associative table," meaning that conceptual retrieval can be based on a specified value in either the left-hand or right-hand column, and that the retrieval may yield multiple values from either column.

If the values in the left-hand column are unique, then the relation is functional. These general relationships are represented by pairs of indexes, one index mapping the left-hand column of elements to the right, and a second index mapping the right column of elements back to the left. By design, virtually all relationships in ADAMS—notably attributes and maps—are functional. Because the underlying relationship is thereby known to be functional, retrievals based on a single key using the left-to-right index always return a single value; the right-to-left index must, in general, return sets of values on a single-key retrieval. This section describes the way that these indexes are managed. The actual structure of an index is irrelevant. One may, for example, use B-trees to implement key directed lookup in the table. ADAMS employs a variant of B-trees known as O-trees [Orl89].

7.1. Basic Structures

All indexes used to implement attributes, maps, and sets inherit their structure from the basic class *_A_index*, which maps keys to uniqueids. The keys are defined by the class *_A_key*, which provides constructors for automatic conversion of strings and uniqueids to keys.

Indexes were designed to take advantage of the object-oriented properties of C++. The basic operations on this class include:

```
void insert ( _A_key&,  _A_uid )
      //  Inserts a (key, uniqueid) tuple into the index.

void chg_val ( _A_key&,  _A_uid )
      //  Changes the uniqueid associated with the key.

void remove ( _A_key&,  _A_uid )
      //  Removes the designated tuple from the index.
      //  For indexes that require keys to be unique, the uniqueid
      //  is left as a null value (using the constant _A_NULLUID).

_A_uid get_val ( _A_key&,  _A_ACCESS )
      //  Returns the uniqueid associated with a unique key.
      //  The _A_ACCESS currently has no function, but may later be
      //  used with the transaction manager.
```

```
int nbr_tuples ()
      //  Returns the number of tuples in the index.

_A_uid save ()
      //  Makes an index persistent, returning the uniqueid to be
      //  associated with that index.  Once an index is persistent,
      //  the destructor ~_A_index() will automatically save any
      //  changes every time that an index goes out of scope.
```

7.2. Maps

Maps are implemented using the *_A_map* class, which accepts tuples of the form (<source_uid>, <target_uid>). Each tuple is stored under a storage uniqueid, which is then referenced in two types of indexes, both maintained by this class:

_A_fwd_map

```
      //  This is the forward map index, which maps the sources to
      //  the storage uniqueids.
      //  This class is derived from the _A_index class, and its
      //  member functions are identical, with the exception that
      //  they take _A_uid's instead of _A_key's as the first
      //  elements of their tuples.
```

_A_inv_map

```
      //  This is the inverse map index, which maps the targets to
      //  the storage uniqueids.
      //  It is also derived from the _A_assoc_tbl class, but differs
      //  in that it allows non-unique keys.
```

Besides *insert*() and *remove*(), member functions for this class include:

```
void chg_key ( _A_uid& old_key,  _A_uid& new_key,  _A_uid& elem_uid)
      //  This function removes the tuple
      //      (source uid old_key, storage uid elem_uid)
      //  and replaces it with
      //      (source uid new_key, storage uid elem_uid).

_A_set* get_vals ( _A_uid&,  _A_ACCESS )
      //  This function takes a target uniqueid and returns a
      //  pointer to a set that contains all the source uniqueids
      //  mapping to the target.
```

7.3. Attribute Functions

Attribute functions are implemented using the *_A_attr* class, which accepts tuples of the form (source uniqueid, attribute value). Each tuple is stored under a storage uniqueid, which is again referenced in two indexes:

_A_fwd_attr

```
      //  This index is derived from _A_index and accepts
      //  (source uniqueid, storage uniqueid) pairs.
      //  The only difference between the member functions for this
      //  class and for its ancestor are that it uses _A_uid's
      //  instead of _A_key's.
```

_A_inv_attr

```
      //  This index is also derived from _A_index, but it accepts
      //  (attribute string value, storage uniqueid) pairs.
```

Its member functions include:

```
void chg_val ( char* old_key,  char* new_key,  _A_uid& elem_uid )
      //  This removes the tuple
      //       (string value old_key, storage uniqueid duid)
      //  and inserts the tuple
      //       (string value new_key, storage uniqueid elem_uid).

_A_set* get_vals  ( char*,  _A_ACCESS )
      //  This function takes an attribute value as an argument and
      //  returns a pointer to a set containing the source element
      //  uniqueids that map to that value.

_A_set* get_range_vals ( char* lo_key,  char* hi_key,  _A_ACCESS )
      //  This function returns a pointer to the set of all source
      //  element uniqueids that map to attribute values that fall
      //  within the specified range.
```

7.4. Index Manager

Virtually, any executable ADAMS program must access at least one index to do any useful work. Currently these indexes are implemented as objects belonging to a number of C++ classes. These classes have constructors for loading a persistent image, destructors for closing persistent images, and methods which may be performed on objects to do work. The preprocessor must generate code to access and manipulate these indexes. To perform operations on an index, the persistent image is loaded into memory resident data structures with C++ constructors. This could be done by declaring an object of the appropriate index class and passing as a parameter the uniqueid of the required index.

_A_set aset ([uniqueid of the index]);

This gives the executing program a handle, aset, by which operations may be performed on the index. By **handles** we mean the storage location of an actual memory resident data structure that represents a set, map or attribute. Conceptually, the executable ADAMS program could access these indexes, opening, closing and performing operations on them via C++ constructors, destructors and methods. However, this presents a number of problems.

First, ADAMS is an embedded language. The concept being that it may be embedded in languages like C, FORTRAN, Pascal and Ada. Code generation from the preprocessors viewpoint is a complex task. The generated code must have a method to access the C++ objects (data structures), which are implemented with C structs, and methods which make up the ADAMS run time system. Embeddability presents problems, in that it is necessary to restrict interaction between ADAMS and a host language, while maintaining a flexible, powerful interface to the ADAMS run time system. This interaction must be well defined. For example, in the code generated by the preprocessor the actual data structure (C struct) that is a set object would need to be declared before being used in the program. Also, if this set were persistent and already contained elements the C function that is the C++ constructor would have to be explicitly called to load the set into memory. Likewise, the destructor would have to be called when the set is no longer needed. These declarations and explicit function calls are messy. Further, if the generated code is forced to make declarations of objects to be used before their use and clean them up when no longer needed, certain host language characteristics may effect the semantics of ADAMS programs.

Second, when ADAMS is embedded in a language like FORTRAN which does not support the concept of a record or a struct, the complexity is multiplied, and strict knowledge of each data structure is needed by the preprocessor. Since, these objects would exist as declared data

structures, then the objects would be governed by the scoping rules of the host language instead of the scoping rules of the ADAMS language causing possibly different semantics for the same ADAMS program depending on the host language. This is an unacceptable level of complexity for the preprocessor.

What is needed is a method for the code generated by the preprocessor to perform operations on ADAMS indexes, without having to maintain handles to the indexes in any language. Our solution is to have this task performed by an *Index Manager*, hereafter referred to as the IM. The IM along with the glue routines (discussed later) serve as the link between an executing ADAMS program and the ADAMS run time system. Even though the IM is part of the final executable program, conceptually it is very separate. The IM maintains all of the handles to ADAMS indexes needed by the executing program. The glue routines use these handles to sets, maps and attributes to perform the operations on them by using the methods defined on these C++ classes.

The glue routines are a very simple interface to the preprocessor. These routines are used in the code generated by the preprocessor to perform the semantic actions specified by ADAMS statements. The glue routines call upon the IM for handles to the objects upon which operations are to be performed. The IM in turn maintains the objects, returning handles to previously accessed indexes and opening new ones when needed. Further, the IM may be told when to flush itself for implementation of transactions.

The preprocessor still needs a way of referring to objects. This is most naturally done by uniqueid. The concept of the uniqueid for ADAMS elements is fundamental to the language as well as the implementation of the runtime system. Furthermore, a uniqueid is easily representable in any language. In retrospect, the preprocessor refers to indexes by uniqueid when calling glue routines. The glue routines perform the function of the C++ class methods. The routines obtain handles to the index from the IM. With this handle the glue routine may then use the C++ methods defined on that class to perform the operation required. This in conjunction with the IM and glue routines removes the problem of complex handle maintenance thus making it much easier to embed ADAMS in almost any language in a timely fashion.

7.5. Index Manager Interface

The idea behind the IM is a simple one. The preprocessor generates the code to perform ADAMS operations via glue routines. Having removed the responsibility of maintaining handles from the preprocessor. The preprocessor refers to indexes by uniqueid. To perform an operation on an index, the preprocessor passes the uniqueid of the index to be used in an operation to the appropriate glue routine. The glue routines interact with the IM and receive handles (pointers) to the objects required. These pointers reference the core image of the appropriate C++ object. Glue routines then utilize the methods defined on these classes to perform the operation returning values to the executing program when necessary. The overall relationships between the host language code, the index glue routines, and the index manager is illustrated in Figure 7.1.

7.6. Index Manager Operations

The internal workings of the IM are as simple as the concept. It only has three operations that may be asked of it:

```
void* get_index (_A_NDX_TYPE type,  _A_uniqueid uid);
void  cleanup_indexes ();
void  close_indexes ();
```

Get_index is used exclusively by the glue routines. *Cleanup_indexes* and *close_indexes* are used in code generated by the preprocessor.

Figure 7.1

(1) *Get_index* takes as parameters the type of the index to be opened (set, map, attr) and the uniqueid of this index. From the view of the glue routines, it does not matter if the index is open or closed. *Get_index* returns a handle to it if it is open, opening it if needed.

(2) *Cleanup_indexes* calls are made at the end of a transactions. The calls are placed by the preprocessor. A cleanup_indexes call closes all persistent (non-local) indexes, thus freeing them for use in other transactions. Non-persistent or local indexes are kept open since they are not used outside of the scope of the ADAMS program and do not matter for concurrency considerations. An entire ADAMS program is treated as a default transaction.

(3) *Close_indexes* is a call made at the end of an ADAMS program. This call closes all indexes, writing updates to disk and removing the disk images of all non-persistent (local) indexes.

7.7. Index Manager Implementation

The IM performs its task with the use of the constructors and destructors defined on the C++ objects. These functions take the uniqueids of objects as parameters and load persistent indexes. The index type is needed so the IM will know which constructor or destructor to call. All IM indexes are maintained on the heap. The IM currently uses a linked list to store handles using a Move to the Front (MTF) heuristic to insure a rough ordering by frequency of use. It is not clear that a tree would be faster but the option should be explored. The MTF heuristic is optimal for iteration through a set where a handle is requested repeatedly. In this case the access will be constant time after the first access. Perhaps an optional implementation would be to maintain a binary tree with a pointer to the last item accessed which would always be checked first for a hit. A test should be made to determine whether accesses to the MTF list are on average better than the $\log n$ lookup of a tree.

This self organizing list is singly linked and made up of nodes of the following type:

```
struct  _A_indexman_tbl_node
      {
      _A_uniqueid                   index_uid;
      _A_NDX_TYPE                   index_type;
      _A_BOOLEAN                    LOCAL;
      void*                         index;
      struct _A_indexman_tbl_node*  next;
      };
```

When a get_index call is made the IM traverses the linked list searching for a uniqueid match. If a match is found the node is moved to the head of the list and the pointer is returned. If no match is found the internal function open_index is called. The appropriate constructor is called and the index is loaded into memory, and added to the front of the linked list. A pointer is then returned to get_index and then the calling glue routine in turn.

It is important to note that although an index may exist in the dictionary, placed there at run time by the code generated by the preprocessor, the index may not have a disk image yet. This fact is totally transparent to all routines and is managed by the IM. Therefore, before a constructor is called, a uniqueid method, *storage_exists()*, is called on the uniqueid which is itself a C++ object. This function returns TRUE or FALSE indicating whether or not the object has been created in the database. If the object has been previously created the constructor is called with the uniqueid as a parameter thus loading the index into memory. If the index has not been created (ie. no disk image has been created) the IM creates an image.

Further, a dictionary call (to _A_check) is made to determine the scope of an index (LOCAL or persistent). In this fashion, the IM knows what can be maintained throughout the course of transactions and programs. When a new index is instantiated the IM must initialize the uniqueid of the index object as well as the persistence level. (LOCAL or persistent) The aforementioned constructor calls on objects are performed via the C++ *new* function. *New* accepts the class of the object as well as any parameters needed by a constructor of that class, or no parameters for uninstantiated indexes. An example call would look like:

```
set_pointer = new  _A_set(a_uniqueid);
```

This call opens the persistent set referred to by the uniqueid *a_uniqueid*.

The IM is not a single entity. Instead, a copy of it exists in every executable ADAMS program. The code and data structures it uses, exist in the file *indexman.h* and *indexman.c*. The IM is compiled separately and linked with every ADAMS program. It will be placed in a library for use when compiling ADAMS programs.

An important implementation note is that void* variables (void pointers) are used to maintain the handles in the IM. This is done because it is easier to cast where needed in the IM and glue routines than to worry about adding extra functions and overhead to the IM.

A very important issue that has not been addressed as of yet is what will happen if a program crashes, or a fatal system error occurs. If this occurs the close_indexes call will never be made. This is not a real problem for persistent indexes since all modifications will be write through, and we will assume the existence of an acceptable roll back mechanism. The problem arises for nonpersistent (local) indexes, their image may never be removed from disk. This problem could normally be managed in the index destructors when the call was made to *close_indexes*. Also, since local indexes are allowed to remain intact across transactions, it may be difficult to have the Transaction Manager handle this problem. A solution may be found in the fact that each ADAMS program is considered a default transaction. In light of this, all writes to stable storage could be rolled back or in this case removed. Appropriate safe guards must be provided, as these indexes may very well begin to eat disk space. Further, in the case of sets if link counts of persistent elements are incremented when these elements are placed in a local set [point for discussion!], it will fall upon the recovery system to decrement these linkcounts in the event of a fatal system error.

7.8. Index Glue Routines

The glue routines operate in conjunction with the Index Manager (IM). They are basically a set of functions which serve as a simple C interface to the runtime system, through which the preprocessor performs the semantic actions of the ADAMS program. As discussed in the IM section, the IM manages handles to the indexes used by an executing program. This creates a stable interface where all actions are performed by calling a glue routine and specifying the uniqueids of the objects to be used in the operation. The glue routines make a get_index call to the IM which returns handles to the required objects. The required operation can then be performed using C++ methods defined on that class of index.

Most of the glue routines are very simple, consisting of perhaps only a few lines of code. Glue routines have been defined for sets, maps, attributes and uniqueids. The source code for all of these routines exists in *indexglue.c*. An important note, is that glue routines are also used for access to other facilities of the runtime system not just indexes. New uniqueids, for example, may be gotten and instantiated. The routines are listed below:

Set Manipulation Routines:

```
void  _A_set_insert ();
void  _A_set_remove ();
int   _A_set_member ();
void  _A_set_union ();
void  _A_set_intersect ();
void  _A_set_complement ();
void  _A_set_assign ();
int   _A_set_first_element ();
int   _A_set_next_element ();
```

Map Manipulation Routines:

```
void  _A_map_insert ();
void  _A_map_remove ();
void  _A_map_change ();
void  _A_map_get_val ();
void  _A_map_get_set ();
```

Attribute Manipulation Routines:

```
void  _A_attr_insert ();
void  _A_attr_assign_value ();
void  _A_attr_remove ();
int   _A_attr_get_val ();
void  _A_attr_get_set ();
```

Uniqueid Manipulation Routines:

```
void  _A_uniqueid_instant ();
void  _A_uniqueid_getuid ();
```

The source code for the routines is relatively simple. Set, map and attribute manipulation routines use the IM and C++ class methods for performing operations. All of the glue routines accept string representations of uniqueids rather than uniqueids, where needed. This completely removes any direct interaction between the code generated by the preprocessor and the C++ classes which make up the run time system. The glue routines take this string representation and use it in the declaration of a uniqueid auto variable. A constructor defined for uniqueids makes the conversion from string to uniqueid, so it may be used. Uniqueids for indexes are used in a function call to get_index (IM), and a local pointer to the appropriate index is declared to hold the returned handle.

Two particularly note worthy glue routines are: *_A_set_union* and *_A_set_intersect.* These routines are a little tricky. When performing either of the functions the preprocessor does some packaging before the call. Both functions get a string representation of the uniqueid of the set to hold the resultant, and an array of string pointers. Each string pointer in the array references the string representation of the uniqueid of one of the sets to be unioned or intersected. A temporary set is opened for building the resultant, because the set specified to hold the resultant set may also be included in the operand list. Then the appropriate augmented assignment operator (+= or *=) is used on each operand set in turn. The set to hold the resultant is then emptied and an assignment is made. In the future changes may be added to perform some of these operations in parallel.

Here is a brief example a glue routine, this routine is for set manipulation:

```
void  _A_set_insert  (char* uidstr, char* setstr)
      //  function:   Performs a set insertion.
      //  parameters: char* uidstr; string rep of elem uid to be inserted.
      //              char* setstr; string rep of the set uid.
      {
                                          // Get the set.
      _A_set*  insertset = (_A_set*) _A_get_index(_A_SET, setstr);
      _A_uniqueid insertuid (uidstr);     // convert string to uniqueid.
      insertset->insert(insertuid);
      };
```

Its operation is relatively simple. The character string representations of the uniqueids of the set and the ADAMS element to be inserted to the set are passed as parameters to the function. The call to _A_get_index is made with the appropriate index type *_A_SET* and the uniqueid of the set as parameters. A handle is returned to the variable insertset which is a pointer to the opened index. Note, the function _A_get_index requires a uniqueid as a parameter, and the parameter passed is actually a string. An implicit call is made to the _A_uniqueid class constructor which converts the string to a uniqueid for use in the call. Similarly, a local variable of type _A_uniqueid is declared and the string for the uniqueid to be inserted into the set is passes as a parameter to an explicit call to the same constructor where a conversion takes place. The returned handle to the set is then used in a call to a set method to perform the insert operation.

8. Sets

Sets are fundamental ADAMS structures. Sets may be thought of as simple aggregate structures with ADAMS elements as members. ADAMS sets are mathematical sets where all elements in a set must be distinct.

8.1. Set Implementation

Sets, Maps and Attributes are currently implemented on a single underlying structure. In the C++ hierarchy of defined classes, sets are derived from *_A_index* class. The *_A_index* basically stores tuples of the form (key, value). (The current implementation is a memory resident, doubly linked, ordered list. Sets use this structure to store (uniqueid, uniqueid) pairs. This has the consequence that sets must always be totally memory resident. This will change with the implementation of O-trees, in that sets will no longer be required to exist entirely in memory in order to perform operations on them.) The ordering of the set is currently performed on the value of the uniqueid. This ordering is hidden and not part of the semantics exported to the final users of the ADAMS language.

8.2. Set Methods

A number of basic methods exist for use with sets. These methods are member functions defined for the C++ class **set**. Their function is relatively straight forward.

Set Manipulation routines:

```
void  insert ( _A_uniqueid& elem_uid );
          //  insert 'elem_uid' into the instance set.
void  remove ( _A_uniqueid& elem_uid );
          //  remove 'elem_uid' from the instance set.
void  add_elements ( _A_set& );
          //  append all elements of 'arg_set' to the instance set.
void  empty ();
           //  remove all elements from the instance set.
```

The functions *insert*, *remove*, *empty* and *add_elements* are used for building and modifying sets. *Insert* and *remove* take an element uniqueid as parameter, and as their name suggests insert and remove the specified element respectively. *Empty* totally removes all elements from a set, and *add_elements* places all elements from one set into another set. The latter is used almost exclusively for a fast shallow copy from one set to an empty set. This is used almost exclusively in the glue routines where the set the operation is to be performed on is known to be empty. It is not a true union operator. Consequently, indiscriminate use of this can violate the rule that set elements are unique within that set because, no check is made to verify that the set upon which the operation is performed is empty for efficiency reasons.

Set Membership routines:

```
_A_BOOLEAN  member ( _A_uniqueid& elem_uid );
              //  test 'elem_uid' for membership in the instance set
```

Set Iteration routines:

```
_A_uniqueid  first_val ( _A_ACCESS acc );
               //  return uniqueid of first element of instance set.
               //  DEFAULT: 'acc' code is WRITE.
_A_uniqueid  next_val ( _A_ACCESS acc );
               //  return uniqueid of next element of instance set.
```

The two functions above are used for iterating through the elements of a set. *First_val* resets an internal pointer to the first element in a set. *Next_val* returns element uniqueids one at a time until the set is exhausted. When this occurs a predefined constant _A_NULLUID is returned.

Again an explicit ordering of the set is not part of set semantics exported to ADAMS users. The only guarantee made is that users may iterate through the contents of a set.

Set Operations:

```
void        operator  +=  (_A_set&);
void        operator  -=  (_A_set&);
void        operator  *=  (_A_set&);
void        operator  =   (_A_set&);
_A_BOOLEAN  operator  ==  (_A_set&);
```

The C++ operators +=, *= and -= are used for set unions, intersections and relative complements respectively. These are C++ methods on the class set, and are performed on a set with another set as a parameter. They are unusual in that the resultant of the set operation is placed in the set upon which the operation is performed. Take as an example, *a += b*; This operation takes the union of the sets *a* and *b* placing the resultant set in *a*. The = and == operator are used for a shallow copy (copy uniqueids only) and a boolean set comparison respectively.

The augmented operators were developed for common set manipulation. This method of performing set unions, intersections and complements was chosen for a specific reason. The reason being that although the +, * and - operators would be more expressive, their use as binary operators result in the generation of auto variables. Take as an example the following code fragment:

```
a = b + c + d;
```

While this is more expressive than:

```
a += b;
a += c;
a += d;
```

the use of binary operators in the first manner generates three auto variables. In other words three sets are generated and destroyed in the execution of the first expression. The other expression generates no auto variables. The generation of a simple auto variable such as an integer presents no problem. Recall, however that sets may contain thousands of elements and will be implemented with possible extensive disk images. The overhead required to generate auto variables of this magnitude is unacceptable. Further use of the ugmented operators with some temporary sets immediately suggests how the operation may be parallelized in the case of unions and intersections over a number of sets.

It is important to note that these set operations and the C++ concepts of classes (not to be confused with ADAMS classes) and objects are hidden from the viewpoint of the code generated by the preprocessor. They make up the ADAMS run time system, and are used exclusively by the run time system and glue routines.

9. Retrieval, Inverse Operators

9.1. Basic Structures

All retrieval mechanisms inherit their structure from the basic class *_A_index*, which maps keys to storage uniqueids. The keys are defined by the class *_A_key*, which provides constructors for automatic conversion of strings and uniqueids to keys. The basic operations on this class include:

```
void insert ( _A_key&,  _A_uniqueid )
      //  Inserts a (key, storage uniqueid) tuple into the table.

void chg_val ( _A_key&,  _A_uniqueid )
      //  Changes the storage uniqueid associated with the key.

void remove ( _A_key&,  _A_uniqueid )
      //  Removes the designated tuple from the table.
      //  For tables that require keys to be unique, the uniqueid is
      //  left as a null value (using the constant _A_NULLUID).

_A_uniqueid get_val ( _A_key&,  _A_ACCESS )
      //  Returns the uniqueid associated with a unique key.
      //  The access currently has no function, but may later be
      //  used with the transaction manager.

int nbr_tuples ()
      //  Returns the number of tuples in the table.

_A_uniqueid save ()
      //  Makes a table persistent, returning the uniqueid to be
      //  associated with that table.  Once a table has been saved,
      //  the destructor ~_A_index() will automataically save any
      //  changes every time that a table goes out of scope.
      //  (The index manager achieves this effect by simply setting
      //  the member variable persistent to _A_TRUE.)
```

9.2. Maps

Maps are implemented using the *_A_map* class, which accepts tuples of the form (source uniqueid, target uniqueid). Each tuple is stored under a storage uniqueid, which is then referenced in two types of tables, both maintained by this class:

_A_fwd_map

```
      //  This is the forward map table, which maps the sources to
      //  the storage uniqueids.
      //  This class is derived from the _A_index class, and its
      //  member functions are identical, with the exception that
      //  they take _A_uniqueid's instead of _A_key's as the first
      //  elements of their tuples.
```

_A_inv_map

```
      //  This is the inverse map table, which maps the targets to
      //  the storage uniqueids.
      //  It is also derived from the _A_index class, but differs
      //  in that it allows non-unique keys.
```

Besides *insert*() and *remove*(), member functions for this class include:

```
void chg_key ( _A_uniqueid& oldkey,  _A_uniqueid& newkey,
```

```
            _A_uniqueid& elem_uid)
      //  This function removes the tuple
      //    (source uid oldkey, storage uid elem_uid)
      //  and replaces it with
      //    (source uid newkey, storage uid elem_uid).

_A_set* get_vals ( _A_uniqueid&,  _A_ACCESS )
      //  This function takes a target uniqueid and returns a
      //  pointer to a set that contains all the source uniqueids
      //  mapping to the target.
```

9.3. Attribute Functions

Attribute functions are implemented using the *_A_attr* class, which accepts tuples of the form (source uniqueid, attribute value). (The "unk" in the name means "unknown," referring to this function's ability to use user-specified functions to convert arbitrary data type values to strings, which are then used as the keys for ordering the table. The user can also specify a function for reversing the conversion. This is currently the only way of specifying fetch and store methods, but it is currently unaccessible from the ADAMS language.) Each tuple is stored under a storage uniqueid, which is again referenced in two tables:

_A_fwd_attr .

```
      //  This table is derived from _A_index and accepts
      //    (source uniqueid, storage uniqueid) pairs.
      //  The only difference between the member functions for
      //  this class and for its ancestor are that it uses
      //  _A_uniqueid's instead of _A_key's.
```

_A_inv_attr .

```
      //  This table is also derived from the _A_index, but it
      //  accepts (attribute string value, storage uniqueid) pairs.
```

Its member functions include:

```
void chg_val ( char* oldkey,  char* newkey,  _A_uniqueid& elem_uid )
      //  This removes the tuple
      //    (string value oldkey, storage uniqueid duid)
      //  and inserts the tuple
      //    (string value newkey, storage uniqueid elem_uid).

_A_set* get_vals ( char*,  _A_ACCESS )
      //  This function takes an attribute value as an argument and
      //  returns a pointer to a set containing the source element
      //  uniqueids that map to that value.

_A_set* get_range_vas ( char* lokey,  char* hikey,  _A_ACCESS )
      //  This function returns a pointer to the set of all source
      //  element uniqueids that map to attribute values that fall
      //  within the specified range.
```

10. Issues of Parallelization

The introduction of parallelism into ADAMS raises a number of thorny issues. Foremost among these is the decision of whether to impose a particular model of computation on users or not. Because we cannot support an unconstrained shared memory model within the ADAMS run-time system without considerable reprogramming effort, we will (at least initially) support only a message passing environment. We need not concern ourselves with the exact particulars of the environment, but in general, we expect to operate in a single-threaded environment with disjoint address spaces.

Our natural unit of concurrency is the transaction. Accordingly, we will support parallelism in the form of concurrent transactions only. ADAMS will make no effort to augment the host language with extensions intended to facilitate the expression of parallelism. In addition, we will endeavor to exploit parallelism in the **for_each** statement by using loop unrolling techniques. Initially, only ADAMS statements will be examined. This will require some analysis by the compiler, perhaps a second pass. The justification here is that we might be able to achieve significant performance gains by prestaging loop arguments from the persistent store.

Initially, we will not seek a parallel implementation of any aspect of the run-time system. We should first shake down the transaction mechanism and concurrency control. After we have gained some confidence in this aspect of ADAMS, we can begin the conversion of the run-time to MPL (Mentat Programming Language). This conversion may require some architectural changes to the run-time, but should not present any real problem especially since we are syntactically compatible with MPL already.

11. Unimplemented Features

The following features of the ADAMS syntax are currently unimplemented, and we expect will remain unimplemented for the forseeable future.

Codomains:

<membership_clause>
<access_method_clause>
<other_codomain_method>
<add_codomain_method>
<undefined_clause>
<unknown_clause>
<subscript_pool_decl_stmt>
<extend_pool_stmt>

Attributes, maps:

<restriction_clause>

Classes:

<clustered_attr_enum>
<predicates>

Sets:

<retrieval_set>

ADAMS names:

<parameterized_names>
<range_subscripts>

12. References

[AgG89] R. Agrawal and N. H. Gehani, ODE (Object Database and Environment): The Language and the Data Model, *Proc. 1989 ACM SIGMOD Conf. 18,* 2 (June 1989), 36-45.

[Bar89] P. Baron, The ADAMS Preprocessor, IPC TR-89-009, Institute for Parallel Computation, Univ. of Virginia, Dec. 1989.

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

[Jan89] S. A. Janet Jr., The ADAMS Storage Management System, IPC TR-89-008, Institute for Parallel Computation, Univ. of Virginia, Aug. 1989.

[Orl89] R. Orlandic, *Design, Analysis and Applications of Compact 0-Complete Trees*, PhD Thesis, Univ. of Virginia, 1989.

[PFG89] J. L. Pfaltz, J. C. French, A. Grimshaw, S. H. Son, P. Baron, S. Janet, A. Kim, C. Klumpp, Y. Lin and L. Loyd, The ADAMS Database Language, IPC TR-89-002, Institute for Parallel Computation, Univ. of Virginia, Feb. 1989.

Table of Contents