NBSIR 88-3763

DATA HANDLING IN THE VERTICAL WORKSTATION

April 21, 1988

By:
Thomas R. Kramer

DATA HANDLING IN THE VERTICAL WORKSTATION OF THE AUTOMATED MANUFACTURING RESEARCH FACILITY AT THE NATIONAL BUREAU OF STANDARDS

Dr. Thomas R. Kramer
Guest Worker, National Bureau of Standards, &
Research Associate, Catholic University

April 21, 1988

Funding for the research reported in this paper was provided to Catholic University under Grant No. 60NANB5D0522 and Grant No. 70NANB7H0716 from the National Bureau of Standards.

Certain commercial equipment and software are identified in this paper in order to adequately specify the experimental facility. Such identification does not imply recommendation or endorsement by the National Bureau of Standards, nor does it imply that the equipment or software identified are necessarily the best available for the purpose.

CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

DATA HANDLING IN THE VERTICAL WORKSTATION OF THE AUTOMATED MANUFACTURING RESEARCH FACILITY AT THE NATIONAL BUREAU OF STANDARDS

I. INTRODUCTION

1. CONTENTS

This paper discusses data handling in the Vertical Workstation (VWS) of the Automated Manufacturing Research Facility (AMRF) at the National Bureau of Standards. The descriptions pertain to the system in use during the summer of 1987. The paper covers those aspects of data handling dealt with by the author, which includes most of the VWS modules and subsystems.

Chapter II discusses data systems and structures used in the VWS: structures in active memory, local disk files, and the AMRF global database.

Chapter III presents three static data structures used in the VWS: features, machining operations, and the tool catalog.

Chapter IV presents several dynamic data structures: the VWS world model, the local database manager world model, graphics data, machining data, data required in executing a machine tool process plan, part designs, workpiece models, and process plans.

Chapter V discusses the local database manager: what it does, its three modes of operation, and its interfaces with the AMRF database, AMRF common memory, and the VWS controller.

Chapter VI discusses the reports system, including the names of report types supported by the local database manager.

Chapter VII presents the VWS subsystem used to generate LISP code automatically for reading and writing reports.

Chapter VIII gives an overview of the software used for data handling in the VWS.

2. AUDIENCE

The paper is intended for people interested in concepts and technical details of the VWS, particularly AMRF personnel who are running the VWS or maintaining or improving the software for the VWS. The paper is also intended for other researchers in automated manufacturing. Knowledge of the computer language LISP is useful but not essential to reading this paper. Detailed documentation of the LISP functions that are involved with the systems described here is being prepared separately.

3. BRIEF VWS DESCRIPTION

The VWS is a computer-integrated automated machining workstation. It includes a control system, a computer-aided design system, an automatic process planning system, and an automatic numerical control code (NC-code) generator. The principal machinery is a machining center (Monarch VMC-75 with a GE2000 controller) and a robot (Unimate 4070 with a Val II controller) to tend the milling center. There is quite a bit of ancillary hardware. The system is controlled from a microcomputer (Sun 3/160 with 6Mb memory, BW monitor). Running in stand-alone mode, it is possible to design and machine a simple metal part within an hour. The VWS may also be run as an integrated part of the AMRF. The workstation is described in more detail in [K&J1].

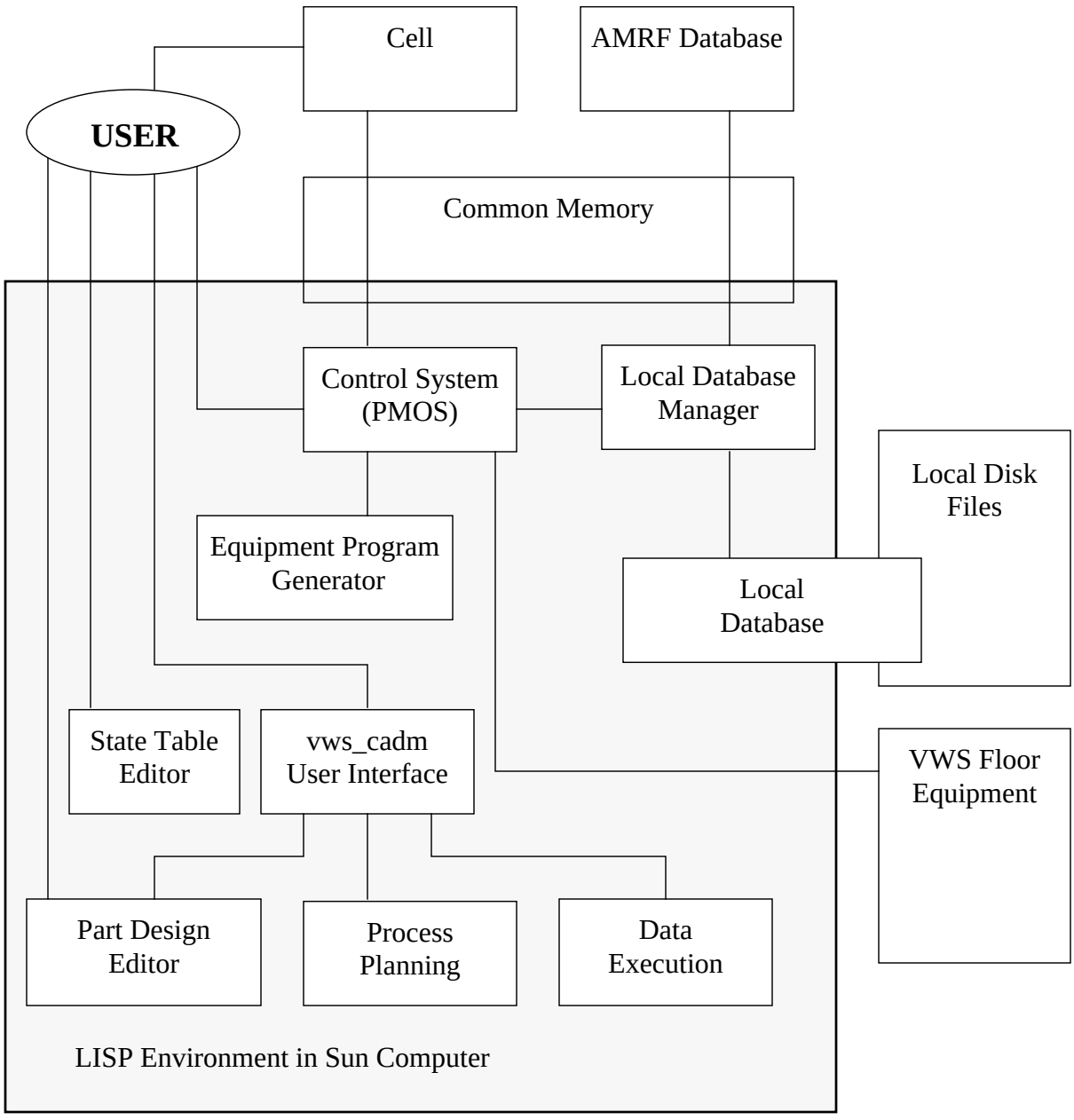
The software for the VWS is written in the Franz LISP dialect of the computer language LISP. In this paper this software is called the VWS2 system. Six principal modules comprise the VWS2 system: the Production Management Operating System (the control system), the State Table Editor, the Equipment Program Generator, the Part Design Editor, the Process Planner, and the Data Execution module. Figure 1 shows the major logical connections of the VWS2 system.

The Part Design Editor, Process Planning and Data Execution modules, as well as other system capabilities, may be accessed by the user through a small user interface called vws_cadm. Vws_cadm asks the user questions about what the user wants to do and then activates the appropriate module or other capability accordingly.

To produce a part from scratch, the user sits at the Sun workstation and creates a design using the Part Design Editor. The Process Planner is then called to write a plan for how to machine a part of that design. Next NC-code is generated automatically from the design and the plan by the Data Execution module. Finally the user tells the control system to make the part. The control system coordinates the activities of the workstation equipment so that a part blank is loaded onto the machining center, the NC-code is sent to the machining center controller and executed (making the part), and the finished part is unloaded.

To make a part using existing data, the VWS controller calls on the local database manager to retrieve several types of data: tray contents, VWS process plan, and NC-code. Following the process plan, the VWS controller directs workstation activity so that the robot loads the part onto the milling machine, the NC-code is sent to the milling machine and executed, and the finished part is unloaded.

Figure 1. VWS2 System Layout



Note: The State Table Editor, Part Design Editor, Process Planning Module, and Data Execution Module all connect with Local Disk Files.

4. RELATED READING

This paper is one of about a dozen papers being prepared as part of the AMRF documentation to describe all aspects of the VWS. The others are [JUN], [KRA1], [KRA3], [KRA4], [K&J2], [K&S2], [KR&W], [LOVE], [NA&J], and [RUDD]. Other papers, prepared for professional meetings, also describe the VWS [KRA2], [K&J1], and [K&S1].

Moving data from one computer to another and between computers and file systems is the function of communications systems. The VWS uses Sun common memory and a network to communicate with the AMRF global database and with the AMRF cell. The common memory software was developed at NBS. The common memory communications system is not discussed in any detail in this paper.

Data handling is the largest single issue in the AMRF and has been discussed in many other papers prepared by AMRF personnel. These include [BAR&], [FUR&], [KRI&], [LIBE], [LI&B], and [RYB&].

II. DATA SYSTEMS AND STRUCTURES

1. INTRODUCTION

Three data storage components are used by the VWS2 system: the working memory of the Sun Computer, disk storage in the file server of the Sun, and the AMRF global database. A directory in disk storage has been designated as the VWS local database. Disk storage also holds the VWS2 software and a great deal of VWS data, such as part designs, process plans, NC-code, and graphics images that may be used at the operator's discretion.

The VWS local database is a surrogate for the AMRF global database. It is used when the VWS is operating in stand-alone mode and when the VWS is operating under control of the AMRF cell but it is desirable to use the local database (such as when the global database is not operating or is overloaded). Interactions between the VWS2 control system and both databases are handled by the VWS2 local database manager.

Computer languages use a computer's on-board memory to store data in various kinds of structures: atoms, arrays, strings, lists, etc. The structures which may be used depend upon the language. Because data is accessed electronically, intricate structures are feasible. Data stored on disk files is always arranged as a linear string of bytes. The disk files, since they are not structured, are often called "flat" files. Throughout the VWS and in the AMRF global database, the bytes represent ASCII characters - with a few exceptions.

There are standard formats in the AMRF for data of different types. The formats are not expressed in any computer programming language. Each application system that uses data from the AMRF global database is welcome to structure data extracted from that database as it pleases for its own internal use. When data is sent to the AMRF global database, it must be sent in a standard format.

Getting data from linear strings of bytes and structuring it in working memory and the reverse operation of extracting data from structures and putting it into appropriately formatted strings of bytes are a large part of the work of local database management. It is normal practice for a computer to be able to read files that describe data structures in the language the computer is currently using and to structure the data in working memory automatically. In the VWS2 system, which is written in LISP, a great deal of data is kept locally in LISP-readable files. This data is trivially transferred into the LISP environment by telling LISP to "load" it. Files in AMRF standard formats, on the other hand, require parsers (special reading programs) for transferring data.

2. LISP DATA STRUCTURES USED

2.1. Overview

LISP provides a range of data types and structures. Unlike many other languages, LISP does not require that data types and structures be defined before they are used. In order to tell what a LISP system is doing, it is necessary to provide a description of the structures being

used as part of the documentation of the system. Providing such descriptions is one of the objectives of this paper.

2.2. Global Variables

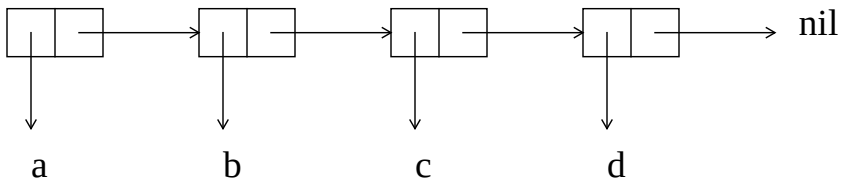
LISP is written mostly as lists of elements, many of which are data constructs called "atoms". Atoms look like ordinary words when they are printed. In lists, some elements are interpreted as the names of functions, some as literals (strings, numbers, and atoms preceded by a single quote) and some as variables. Variables have a value, and are evaluated by LISP when they are used. The value may be local to a function, or global to the entire LISP environment. There are no global variables in the LISP code written by the author for the VWS2 system or under editorial control of the author (Process Planning module, Data Execution module, graphics system, verification system, geometry library, local database manager, and vws_cadm user interface). Global variables are present in other parts of the VWS2 system.

2.3. Lists

LISP is designed to use lists effectively. The VWS2 system makes heavy use of lists. A list of the first four letters of the alphabet would be printed as: (a b c d).

In the computer, a LISP list is represented by "list cells". Each list cell has two parts: the first contains a pointer to the value of the list cell, and the second contains a pointer to the next cell in the list. At the end of a list the second pointer points to the atom "nil". Figure 2 shows the list cell structure of the list (a b c d).

Figure 2. List Structure



When the list (a b c d) is encountered in a LISP-readable file, LISP will automatically set up the internal data structure shown in Figure 2.

The value pointed to by the first part of a list cell may itself be a list.

2.4. Property Lists

One data structure native to LISP is a type of list called a "property list". Each atom in a LISP environment may have a property list in the same way that it may have a value or a

function definition. A property list is assigned to an atom by a simple LISP command. For example, to make (a b c d) the property list of the atom “i_am_an_atom” the following command is issued:

(setplist ’i_am_an_atom ’(a b c d))

Any list may be assigned as the property list of an atom, but to use the property list effectively, it must have a certain format - namely, it must have an even number of elements, and the odd-numbered elements must be atoms.

The idea of a property list is that the odd-numbered elements are the names of properties of the atom, and the even numbered elements are the values of the preceding properties. For example, the property list of a tool named “drill2” might be assigned by:

(setplist ’drill2 ’(tool_type drill length 4.0 diameter 0.5))

Information can be extracted from property lists by the function “get”. For example, to obtain the length of drill2, the following command would be issued:

(get ’drill2 ’length)

The value returned by LISP from that command would be 4.0.

Conceptually, it is useful to think of a property list as being divided into property_name - property_value pairs, but the actual structure and print representation of a property list do not have any explicit pairing.

In LISP, the property list of an atom is a global attribute (unlike the value of a variable, which may be set locally, globally, or both). The VWS2 system does make use of the property lists of atoms. The property lists of the following atoms are reserved in the system: db_mgr_world, drawp, features, machine_ops, mockup, mtool, pde, tool_catalog, vws_world. The property lists of those atoms should not be used for any purposes other than those for which they are set up in the system.

In addition, every design, workpiece, and process plan brought into the system (there may be none or many) is represented as the property list of an atom.

2.5. Hierarchical Property Lists

2.5.1. Structure

The value of a property in a property list may be a list. For example, information about the tip of drill2 might be put into its property list as shown in Figure 3. This data structure is a tree. Figure 3 shows: A. the LISP command to create the list, displaying the printed version of the list, B. the conceptual tree, and C. the internal LISP structure of the tree.

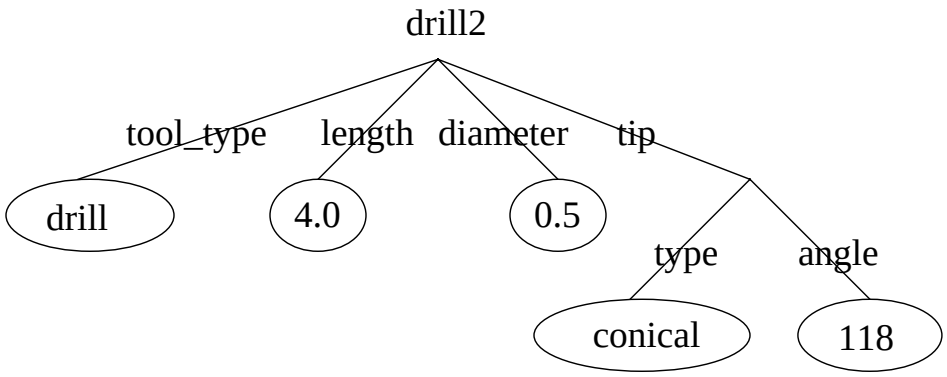
Because the internal LISP structure of the tree is made up of one-way pointers, it is possible to follow the conceptual tree downwards only in LISP (unless extra pointers are added which point upwards).

Figure 3. Hierarchical Property List

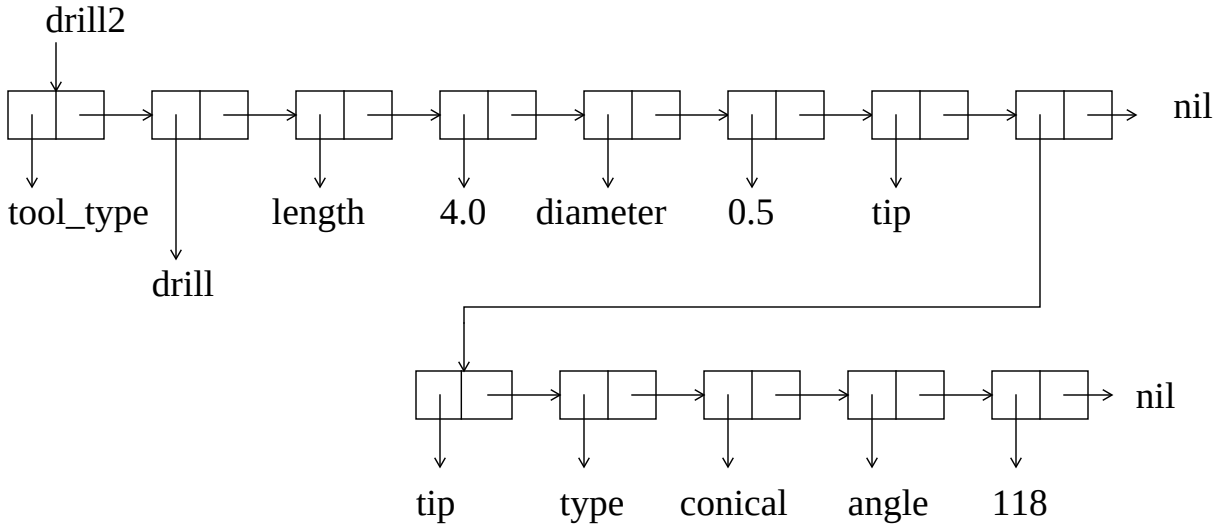
A. LISP Command to Create the List

(setplist 'drill2 '(tool_type drill length 4.0 diameter 0.5 tip (tip type conical angle 118)))

B. Conceptual View



C. List Structure



To get information about the tip of drill2, the following command would be issued:

```
(get 'drill2 'tip)
```

and LISP would return the value (tip type conical angle 118).

If the value of a property is a list, the list itself may (or may not) be usable as a property list. To be usable as a property list, it must have an ODD number of elements, and the even-numbered elements should be atoms. The first element in the list is only a place-holder. It is neither a property nor a value. The rest of the list is a set of property_name - property_value pairs.

In the VWS2 system, the convention has been adopted that if the value of a property is a property list, the first element of the list (the one that is only a place-holder) must be the same as the name of the property. This has been done in the example above, where "tip" is both the name of the property and the first element of the list which is the value of the property. This convention has the name of the property carried in its value, which is often convenient. It also serves as a marker for error checking.

2.5.2. Accessing Routines

The inconsistency in LISP between top-level property lists (which must have an even number of elements) and lower-level property lists (which must have an odd number of elements) presents minor programming difficulties, but does not complicate accessing routines since the LISP accessing routines "get" and "putprop" will work if the value of the first argument is a list with an odd number of elements, as well as if the value is an atom.

Functions named "place" and "fetch" have been written for the VWS2 system to put data into hierarchical property lists and get it back out efficiently.

Fetch is like an extended version of "get". Where "get" will only get the value of a branch from the next layer down, "fetch" will go down through any number of layers. For example, to get the tip angle of drill2 from the data structure set up in section 2.5.1, the following function call is made:

```
(fetch 'drill2 'tip 'angle)
```

This will return the answer 118. On Figure 3b, this function call picks the rightmost branch in the first layer of the tree (labelled "tip") and then picks the rightmost branch in the second layer (labelled "angle").

The "fetch" function will take any number of arguments. The first argument is the name of an atom or the name of a list, and the rest of the arguments are the names of branches, each of which is a sub-branch of the branch named by the preceding argument. If any of the branches named in the arguments does not exist in the tree, the value "nil" is returned. If only one argument is given to "fetch", it will return the value of that argument.

A second function, “place”, will build or revise trees. “Place” takes at least three arguments, the first of which is the name of an atom or the name of a list, and the next of which represent a path down a series of branches. The last argument of place, however, is the value to be placed at the end of the series of branches. If some or all of the branches already exist, “place” follows them. If any new branches are named, “place” builds them. The last argument of place (the value being placed) may be any type of data. If the last argument is a list, a copy of it will be used as the value. This prevents unwanted links between lists from being formed.

Every time it is used, “place” makes the following checks:

A. that none of the branches is named “nil”,

B. that the first atom of any named branch which is a list is the same as the name of the branch,

C. that all of the middle arguments are atoms, and

D. that there are at least three arguments.

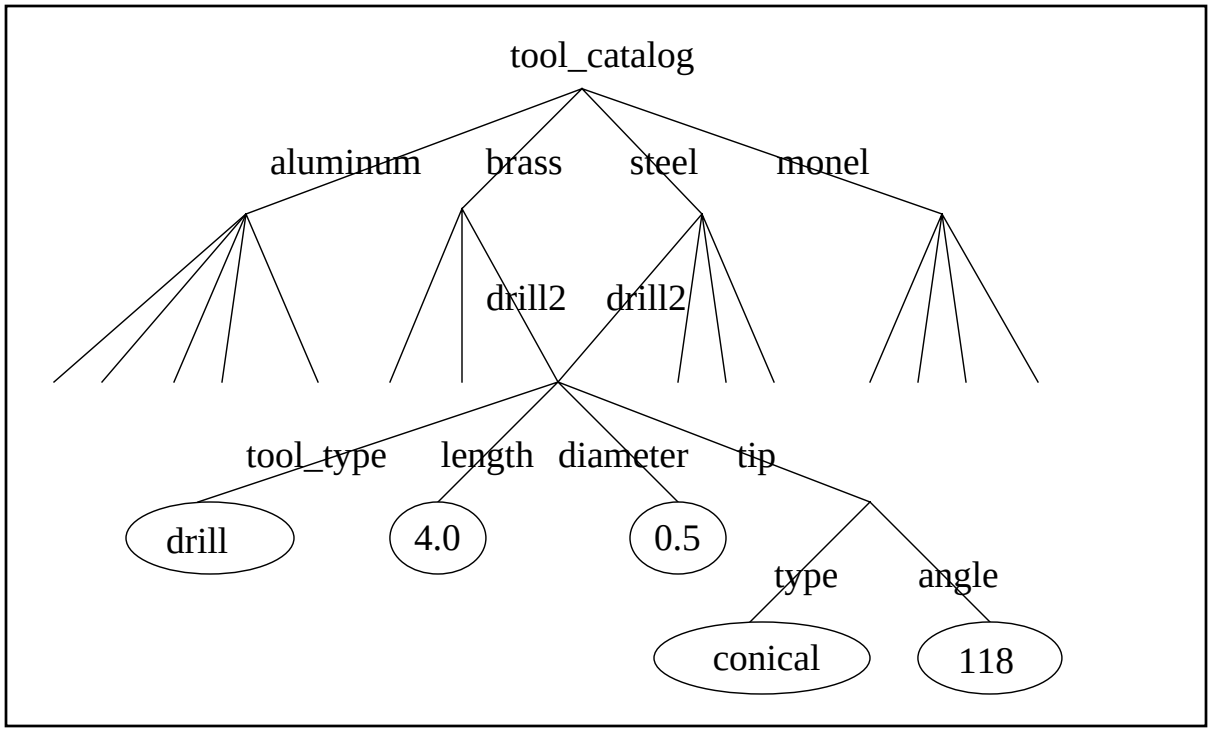
If any of these checks fails, “place” prints an error message describing the problem.

A third function, “putlink”, is identical to “place” except that if the value is a list, it is used without being copied. “Putlink” is used when multiple pointers to the same data are desired.

2.5.3. Linking

For some purposes it is useful to have access to the same data through more than one route. For example, the VWS2 tool catalog contains a section for each of four materials (aluminum, brass, steel, and monel). Each section includes all tools which are suitable to cut the material. In most cases, a given tool will cut several materials. Rather than include the same information several times, or create an indexing system, the tool catalog data structure is built with several pointers to information about a tool, one pointer from each material the tool will cut. A simplified tree showing the linking if drill 2 is suitable for both brass and steel is depicted in Figure 4. The actual tool catalog structure, described in Chapter II, section 4 of this paper, is many times larger and has several more layers.

Figure 4. Linked Hierarchical Property List



There is no LISP-readable version of a linked list of the sort shown in Figure 4. If the list is printed out, it appears to have two copies of the information about drill2, and if the printed version is read back in, two copies of the data will be stored. To create the linked structure, an unlinked version of the data is read in and then linked together with function calls to “putlink” or “putprop”.

If the information about a tool is changed, all routes of access to the information will get the changed version.

2.5.4. Use in the VWS2 System

Hierarchical property lists are the principal data structures used in all the parts of the VWS2 system written or under editorial control of the author. In a few cases (explicitly identified later in this paper) the lists are linked.

2.5.5. Pros and Cons of Hierarchical Property Lists

Hierarchical property lists are extremely convenient for the following reasons:

A. Data may be structured in the same way it is thought about.

B. Raw data in LISP-readable form is very easy for a human to read and comprehend when it is "pretty-printed" (and the pretty-printed format is still LISP-readable). A pretty-printing function for property lists named "pp_plist" is part of the VWS2 system. In pretty-printed format, each layer of the hierarchy is indented further than the preceding layer and the value of a property is printed alongside the name of the property. For example, the pretty-printed version of the property list of drill2 set up in Figure 3 is as follows.

```
(tool_type   drill
 length      4.0
 diameter    0.5
 tip         (tip
              type      conical
              angle     118))
```

C. The data structure may be altered at any time by adding or deleting properties at any level of the hierarchy without requiring changes to data storage and accessing functions. Moreover, the order of the data in the structure itself does not have to be controlled.

D. Because the structure is hierarchical, data accessing is reasonably efficient. Moreover, if the hierarchy grows a large branch, it will make no difference to the efficiency of accessing other branches.

E. It is feasible to write fairly simple functions to restructure data for special purposes. For example, the VWS2 system makes many calls to "pull_prop" and "pull_values", which, respectively, extract all the property names or all the property values from the top level of a property list.

Because finding data in a hierarchical property list requires searching, data access cannot be as fast as if, for example, an item of data were in a fixed position in a list. But speed of data access from hierarchical property lists appears to have been very fast - fast enough that the user interactive parts of the VWS2 system all have response times comfortable to the user, and the non-user interactive parts all complete work within a few minutes on typical jobs.

In some cases the use of properties is artificial; for example, in a list of names of machining operations. It is usually convenient to use the positive integers as property names in such cases. This may create the false impression that there is a natural ordering. The positive integers are also useful as property names when a property is required for identification purposes, but there is no natural method of assigning property names. The numbering of features in a design is an example of this. The positive integers suffer from the drawback that their use depends upon the dialect of LISP. Franz LISP allows the use of integers from -1024 to +1023 as property names. Integers outside that range are not usable.

Some of the uses of hierarchical property lists could be handled using LISP "flavors". Indeed, the VWS equipment program generator described in [NA&J] makes use of flavors. The inheritance and message passing mechanisms of flavors may offer some advantages, but Dr. Jun reports that flavors is not as flexible, flavor-based programs are harder to debug, and the internal data structures of flavors are not accessible.

3. DIRECTORIES AND FILES

3.1. Current Directory

The VWS2 system is written in LISP, but runs under a UNIX operating system on the Sun computer. UNIX maintains files in a hierarchical system of directories, and processes run "in a directory" in the sense that some directory is designated to receive output files from the process and provide input files. The VWS2 Process Planning and Data Execution modules write files in the current directory, whatever that may be. Normally the current directory is either ~kramer/vws2 (which is the author's top-level VWS2 system directory), or ~jun/vws3 (which is Dr. Jau-Shi Jun's top-level directory for the VWS2 system). The VWS control system is normally run using ~jun/vws3 as the current directory.

The four types of files that may be written by the system in the current directory are: process plans, enhanced process plans, NC-programs, and workpiece descriptions. The first of these is prepared by the Process Planning module. The other three are prepared by the Data Execution module.

If it is desired to have the local database manager use any files created in the current directory, the files must be moved (by a simple UNIX command) to the "database" subdirectory of ~jun/vws3.

3.2. Database Directory

The local database manager works for the VWS control system and uses the "database" subdirectory of ~jun/vws3. The local database manager expects to find files it needs in that directory and uses that directory to save files containing NC-programs and process plans obtained from reports provided by the AMRF global database.

When the database manager is running locally or in the "monitor" mode (described later in this paper), it maintains a log of all database transactions in a file called "db_mgr_log". This file is written in the "database" subdirectory.

3.3. Design Directory

The Part Design Editor module, described in detail in [K&J2], normally runs in either ~kramer/vws2 or ~jun/vws3. The module requires a font picture that is available in both of those directories. The module also requires that the current directory have a subdirectory named "design". Part design files are written to that subdirectory and retrieved from it by the module. The vws_cadm user interface also requires that the "design" directory exist when designs are to be obtained through vws_cadm.

4. AMRF GLOBAL DATABASE

The third principal source of data used by the VWS system is the AMRF global database. The AMRF global database is a repository of information about the AMRF. It is described in [BAR&], [FUR&], [KRI&] and [LI&B]. The AMRF global database is used by the VWS through a set of four "mailboxes", which are areas of memory in the LISP environment through which information may be passed into or out of the environment. The mailboxes are for:

A. Commands going from the VWS to the AMRF global database,

B. Status messages (regarding the actions taken to carry out a command) from the AMRF global database to the VWS,

C. Data going out of the VWS to the AMRF global database, and

D. Data coming into the VWS from the AMRF global database.

A communications system which includes a common memory subsystem is required to implement the mailboxes. The communications system is not discussed in any detail in this paper. See [RYB&] for a discussion of AMRF network communications.

Commands and status messages must have certain formats, which are discussed later. Incoming and outgoing data is contained in "reports". In [PO&M], 33 report types are defined. The VWS2 local database manager is ready to handle 10 of these, and a system for enabling the VWS to deal with additional report types is fully implemented. The VWS control system is currently using 3 report types to carry out its work.

The VWS also communicates with the AMRF Cell Controller when the VWS is running in the integrated mode. Additional data is provided along with commands from the cell. This paper does not discuss the cell. See [JUN] for a description of interactions between the VWS and the cell.

III. STATIC DATABASES

1. INTRODUCTION

Static databases are those which do not change during system operation. This chapter presents three static databases used in the VWS2 system: features, machining operations, and tool catalog. All three are kept as hierarchical property lists. Chapter VIII explains how the lists are built in the LISP environment.

In addition to the three described here, the Part Design Editor uses the property list of "pde". This is a one-level deep (not hierarchical) list in which the property names are the names of commands and the values are segments of LISP code to be executed to carry out the commands. The commands are described in [K&J2], Chapter III, section 5.

2. FEATURES

The features database contains information about the part design features used in the VWS2 system. The structure of this database is shown in Table 1 and Table 2.

As shown in Table 1, the features database has six substructures:

A. primary_names - a list of the names of the nine primary features,

B. subfeature_names - a list of the names of the four subfeatures,

C. primary_features - detailed data about each feature type,

D. subfeatures - detailed data about each subfeature type,

E. rule_frags - an equivalence table of LISP variables and phrases which may be used in feature verification rules about any type of feature, and

F. let_end - small sections of LISP code for declaring and setting the values of variables usable in feature verifiers for any feature type.

Table 1 is abbreviated. An example of data that is not shown in Table 1 is given in Table 2, which has detailed data for one primary feature type: pocket. The data for other primary features is structured in the same way as the data for the pocket feature type. As shown in Table 2, the detailed data for a feature type has 13 subsections:

A. rule_frags - an equivalence table of LISP variables and phrases which may be used in feature verification rules about the feature type,

B. set_body -small sections of LISP code for setting the values of variables usable in a feature verifier for the feature type,

C. let_start - small sections of LISP code for declaring variables usable in a feature verifier for the feature type and setting the value of some,

D. req_parms - a list of the required parameters for a feature of the given type. With each parameter name there is a list of type_tests which the value of the parameter must pass in the data for a feature of that type in an enhanced version of a part design.

E. opt_parms - a list of optional parameters for a feature of the given type, also including type tests.

F. subfeatures - a list of the subfeatures that the feature type may have, along with a list of the parameters needed to specify the subfeature.

G. ref_up - a list of the features types which may be reference features for the given feature type, including the name of a LISP function which may be used to determine if the outline of a feature of the given type fits inside the outline of the reference feature.

H. ref_down - a list of the feature types which may have a feature of the given type as a reference feature.

I. verifier - the name of a LISP function which is the feature verifier for a feature of the given type.

J. enhancer - the name of a LISP function which will enhance a feature of the given type.

K. draw_func - the name of a LISP function which will draw a feature of the given type.

L. draw_parms - an ordered list of the names of the parameters needed by the drawing function in order to do the drawing.

M. oper_finder - the name of a LISP function which will select a set of machining operations which will machine the feature.

The data for subfeatures omitted from Table 1 is very similar to the data for primary features. The “rule_frags” and “let_end” data omitted from Table 1 are very similar to the “rule_frags” and “let_start” data shown in Table 2.

Table 1. Features Database

This table shows the structure of the features database. The table is formatted like a LISP command that would set up the database. Omissions of data are indicated by "...". An example of the substructure for one feature (pocket) is shown in Table 2.

```
(setplist 'features '(
primary_names (primary_names
   1  chamfer_out
   2  contour_groove
   3  contour_pocket
   4  groove
   5  hole
   6  pocket
   7  side_contour
   8  straight_groove
   9  text)

subfeature_names (subfeature_names
   1  chamfer_in
   2  chamfer_out
   3  countersink
   4  thread)
```

```
   primary_features   (primary_features
      chamfer_out ...
      contour_groove ...
      contour_pocket ...
      groove ...
      hole ...
      pocket ...
      side_contour ...
      straight_groove ...
      text ...
   subfeatures ...
      chamfer_in ...
      chamfer_out ...
      countersink ...
      thread ...
   rule_frags ...
   let_end ...))
```

Table 2. Features Database - Pocket Data

```
(pocket
 rule_frags (rule_frags
   center_x ((the x_value of the center of the pocket))
   center_y ((the y_value of the center of the pocket))
   wid ((the width of the pocket))
   lenk ((the length of the pocket))
   corner_radius ((the corner radius of the pocket))
   max_rad ((the maximum corner radius of the pocket))
   chamfer_in_depth ((the chamfer_in_depth of the pocket))
   given_depth ((the given depth of the pocket))
   total_depth ((the total depth of the pocket))
   vertical_rise ((the vertical rise of the pocket))
   max_lrx
      ((the x_value of the plus_x side of the pocket))
   max_ulx
      ((the x_value of the minus_x side of the pocket))
   max_uly
      ((the y_value of the plus_y side of the pocket))
   max_lry
      ((the y_value of the minus_y side of the pocket))
   top ((the z_value of the top of the pocket))
   bottom ((the z_value of the bottom of the pocket)))
 set_body (setq
   max_rad (plus corner_radius chamfer_in_depth)
   lenk (or lenk (diff lrx ulx))
   wid (or wid (diff uly lry))
   total_depth (diff given_depth z_surf)
   vertical_rise (min given_depth (plus height z_surf))
   max_lrx (plus lrx chamfer_in_depth)
   max_ulx (diff ulx chamfer_in_depth)
   max_uly (plus uly chamfer_in_depth)
   max_lry (diff lry chamfer_in_depth)
   top (plus height z_surf)
   bottom (diff height total_depth))
 let_start ((center_x (get desc 'center_x)))
   (center_y (get desc 'center_y))
   (ulx (get desc 'upper_l_x))
   (uly (get desc 'upper_l_y))
   (lrx (get desc 'lower_r_x))
   (lry (get desc 'lower_r_y))
   (lenk (get desc 'length))
   (wid (get desc 'width))
   (corner_radius (get desc 'corner_radius))
   (chamfer_in_depth (or (get desc 'chamfer_in_depth) 0.0))
   (given_depth (get desc 'depth))
   (z_surf (get desc 'z_surf))
   total_depth vertical_rise max_rad max_lrx top
   max_ulx max_uly max_lry bottom)
 req_parms (req_parms
   upper_l_x (upper_l_x
      type_tests (type_tests 1 numberp))
   upper_l_y (upper_l_y
      type_tests (type_tests 1 numberp))
   lower_r_x (lower_r_x
      type_tests (type_tests 1 numberp))
   lower_r_y (lower_r_y
      type_tests (type_tests 1 numberp))
   corner_radius (corner_radius
      type_tests (type_tests 1 numberp))
   depth (depth
      type_tests (type_tests 1 numberp 2 plusp)))
 opt_parms (opt_parms
   reference_feature (reference_feature
      type_tests (type_tests 1 fixp 2 plusp))
   chamfer_in_depth (chamfer_in_depth
      type_tests (type_tests 1 numberp 2 plusp)))
 subfeatures (subfeatures
   chamfer_in (chamfer_in
      parms (parms 1 chamfer_in_depth)))
 ref_up (ref_up
   hole (hole ref_test ref_pocket_pocket)
   pocket (pocket ref_test ref_pocket_pocket)
   groove (groove ref_test ref_in_groove)
   contour_groove
      (contour_groove ref_test ref_in_cg)
   side_contour
      (side_contour ref_test ref_in_sc)
   contour_pocket
      (contour_pocket ref_test ref_in_out)
   straight_groove
      (straight_groove ref_test ref_in_out))
 ref_down (ref_down
   1  contour_groove
   2  contour_pocket
   3  groove
   4  hole
   5  pocket
   6  straight_groove
   7  text)
 verifier verify_pocket
 enhancer enhance_pocket
 draw_func draw_pocket
 draw_parms (upper_l_x upper_l_y lower_r_x
   lower_r_y depth corner_radius z_surf)
 oper_finder find_pocket_ops)
```

3. MACHINING OPERATIONS

3.1. Introduction

The machine_ops database contains information about 19 of the 21 machining operations that can be carried out in the VWS. The two which are not included are “init_nc” and “close_nc”, which are the first and last steps of any VWS milling machine process plan. This database is used only by the “execute_step” function, which carries out one step from a process plan. A detailed description of execute_step and how it uses the machine_ops database is given in [KR&W], Chapter II, section 3.

The structure of the machine_ops database is shown in Table 3. The top layer of branches in the database is the 19 operations. Detailed data is shown for only 4 of the 19 operations:

A. drill_hole, which makes a primary feature,

B. machine_chamfer_in, which makes a subfeature,

C. face_mill, which performs a milling operation not related to a feature, and

D. set0_center, which is a zero-setting operation.

The branches of machine_ops for the other 15 operations are all similar to one or another of these four.

Each branch has 9 sub-branches: 4 for verification, 3 for NC-coding, and 2 for drawing. In the six branches which have lists for values, the order of the elements in the list is important, since the lists are used to assemble arguments to LISP functions. The branches whose value is nil could have been omitted from the database entirely without changing the use of the database. They are included with nil values to make it clear that they were not accidently omitted.

3.2. Verification

The VWS2 system is able to make a number of checks on each machining operation, to be sure the operation is safe to perform and will accomplish the intended purpose. This is described in detail in Chapter VI of [K&S2]. In order to accomplish this verification, a great deal of software and data is needed. The four branches of the machine_ops database which deal with verification point to this software and data.

The “test_function” branch gives the name of the verification function used to verify the operation.

The “verf_main_parms” branch is a list of the names of parameters needed by the verification function, whose values are the values of local variables in the “execute_step” function.

The “verf_feat_parms” branch is a list of the names of parameters needed by the verification function, whose values are to be extracted from the data describing the feature in the design on which the operation is working.

The “verf_step_parms” branch is a list of the names of parameters needed by the verification function, whose values are to be extracted from the data describing the step of the process plan being executed.

3.3. NC-Coding

The “nc_function” branch gives the name the function that writes NC-code to carry out the operation.

The “nc_design_parms” branch is a list of the names of parameters needed by the NC-coding function, whose values are to be extracted from the data describing the feature in the design on which the operation is working.

The “nc_step_parms” branch is a list of the names of parameters needed by the NC-coding function, whose values are to be extracted from the data describing the step of the process plan being executed.

3.4. Drawing

The “draw_function” branch gives the name the function that draws the changes in the workpiece which result when the operation is carried out.

The “draw_design_parms” branch is a list of the names of parameters needed by the drawing function, whose values are to be extracted from the data describing the feature in the design on which the operation is working.

The drawing functions never extract any data from the step being executed, so no branch is required in the machine_ops database.

Some operations, such as set0_center, which does not change the workpiece, are not drawn. The value for the two drawing branches for these operations is always nil.

Table 3. Machine-ops Database

```
(setplist 'machine_ops '(
center_drill ...

counterbore ...

drill_hole (drill_hole
  verf_main_parms
    (design feature_id workpiece material)
  verf_feat_parms
    (center_x center_y depth z_surf)
  verf_step_parms
    (changer_slot speed feed_rate pass_depth)
  nc_step_parms
    (changer_slot speed feed_rate pass_depth)
  nc_design_parms (center_x center_y depth z_surf)
  nc_function hole_nc
  draw_design_parms
    (center_x center_y depth z_surf)
  draw_function draw_hole
  test_function drill_hole_test)

face_mill (face_mill
  verf_main_parms (workpiece material)
  verf_feat_parms nil
  verf_step_parms
    (changer_slot speed feed_rate pass_depth
      stepover upper_l_x upper_l_y lower_r_x
      lower_r_y depth z_surf)
  nc_step_parms
    (changer_slot speed feed_rate pass_depth
      stepover upper_l_x upper_l_y lower_r_x
      lower_r_y depth z_surf)
  nc_design_parms nil
  nc_function face_mill_nc
  draw_design_parms nil
  draw_function nil
  test_function face_mill_test)

fly_cut ...

machine_chamfer_in (machine_chamfer_in
  verf_feat_parms
    (chin_ulx chin_uly chin_lrx chin_lry
      chamfer_in_depth chin_cr z_surf)
  verf_main_parms
    (design feature_id workpiece material)
  verf_step_parms  (changer_slot speed feed_rate)
  nc_step_parms  (changer_slot speed feed_rate)
  nc_design_parms
    (chin_ulx chin_uly chin_lrx chin_lry
      chamfer_in_depth chin_cr z_surf)
  nc_function chamfer_in_nc
  draw_design_parms
    (chin_ulx chin_uly chin_lrx chin_lry
      chamfer_in_depth chin_cr z_surf)
  draw_function  draw_chamf_in
  test_function  machine_chin_test)

machine_chamfer_out ...

machine_countersink ...

mill_contour_groove ...

mill_contour_pocket ...

mill_groove ...

mill_pocket ...

mill_side_contour ...

mill_straight_groove ...

mill_text ...

set0_center (set0_center
  verf_main_parms  nil
  verf_feat_parms  nil
  verf_step_parms
    (changer_slot  near_diam near_x near_y
      x_offset y_offset)
  nc_step_parms
    (changer_slot  near_diam near_x near_y
      x_offset y_offset)
  nc_design_parms  nil
  nc_function set0_center_nc
  draw_design_parms  nil
  draw_function  nil
  test_function  set0_center_test)

set0_corner ...

set0_z ...

tap_thread ...))
```

4. TOOL CATALOG

The tool catalog is intended to be a complete list of all the different kinds of tools which can be used for machining in the VWS2 system. The tool_catalog database is a linked hierarchical property list structured as shown in Figure 5. It has two main sections: names and material.

The “names” branch contains property names which are the tool_type_ids’s of tools. Tool_type_ids’s are names of kinds of tools and not identifiers of individual tools. Two of the 48 tool_type_id’s included in the catalog are shown in the figure: drill_0.25_2_abs and tap_0.375_4_abs. The data for these two tools is highlighted by the gray boxes on the figure, the drill in the upper box, the tap in the lower one.

There may be several tools in the VWS tool crib with the tool_type_id “drill_0.25_2_abs”. Each would be uniquely identifiable (by a parameter called “tool_id” not included in the tool catalog), but each would fit the description given in the tool catalog for a tool with the tool_type_id “drill_0.25_2_abs”.

For each tool, there are normally at least seven properties:

A. type - ball_nosed_end_mill, center_drill, chamfer, countersink, drill, end_mill, face_mill, fly_cutter, probe, or tap.

B. cutting_diameter - the diameter of the widest cutting part of the tool.

C. flutes - the number of flutes on the tool.

D. shank_diameter - the diameter of the widest part of the shank of the tool.

E. cutting_depth - the maximum depth the tool can cut.

F. tip - the nature of the tip. For conical tips, this includes tip_type and tip_angle. For other tip types, this includes only tip_type.

G. materials - a list of the names of the materials the tool can cut.

For taps, there is an eighth property: threads_per_inch.

The “material” branch of the tool_catalog has four branches: aluminum, steel, brass, and monel. Figure 5 shows the substructure of the brass branch. The substructure of the branches for the other three materials is the same. The brass branch has one branch: types, which has ten branches, named for the ten tool types being used. Nine tool types have one branch: diameters, which has as many branches as there are different diameter tools of that type which can cut the given material. The tenth tool type, tap, requires an extra subdivision: threads_per_inch, the branches of which are structured like the other nine tool type branches.

The tool_catalog is used by the VWS2 process planning module. It is linked so that it may be used efficiently to retrieve data about tools in two ways:

A. Given the tool_type_id of a tool, get the description of it.

B. Given a material, a tool type, and a diameter, find the tool_type_id of a tool in the catalog of the given type and diameter that may be used to cut the given material.

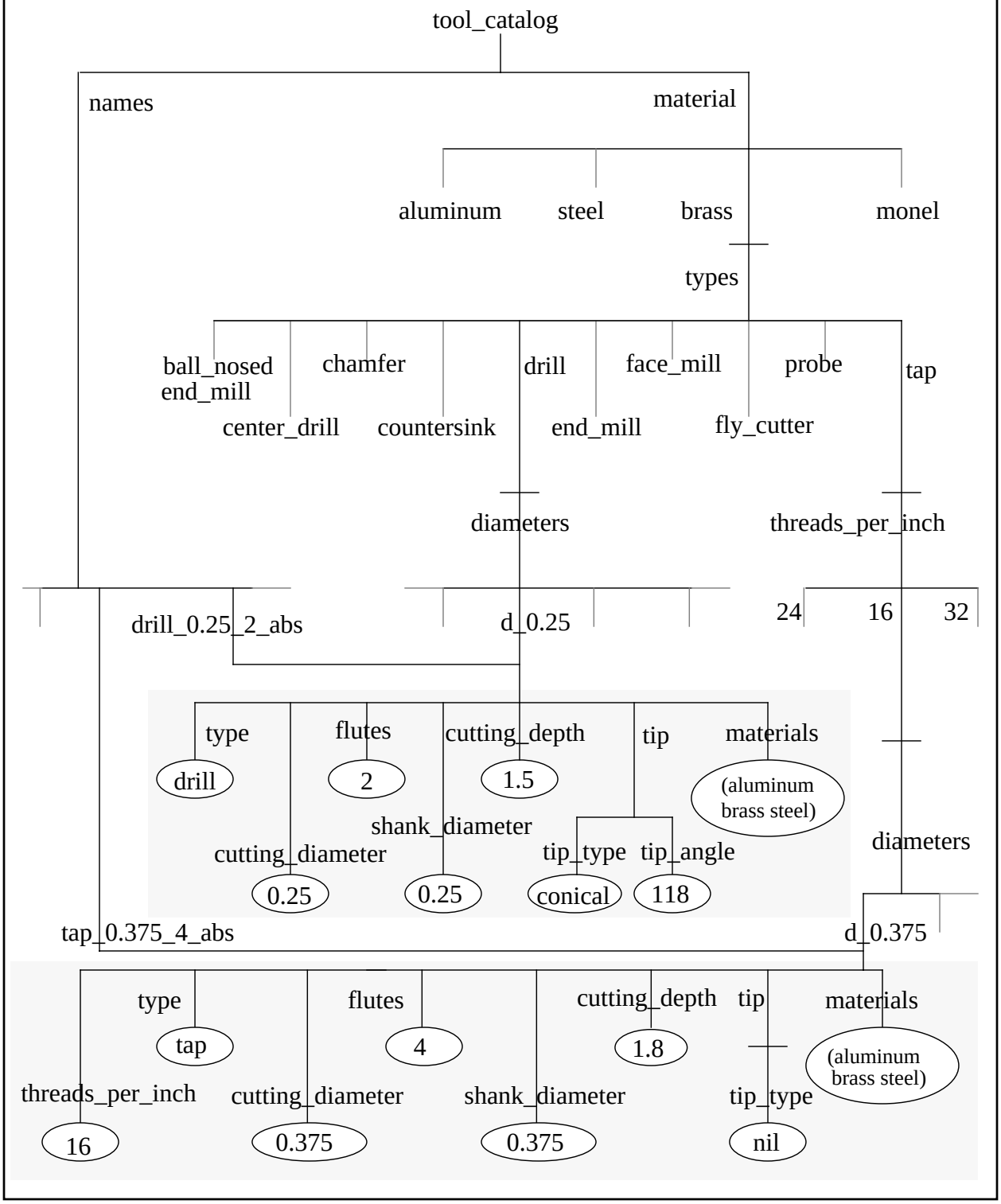
The linking of the tool catalog is done when the VWS2 LISP environment is built. The linking process is totally data-driven by data extracted from the “names” branch of the structure. When a new entry is put in the names branch of the tool catalog, for a tool of a new variety that cuts a new material, and the “init_tools” function is called, the linking process will create all the structure required in the “materials” branch of the tool catalog.

Because the tool_catalog is linked, the following two LISP function calls obtain identical information:

```
(fetch ’tool_catalog ’names ’drill_0.25_2_abs)
(fetch ’tool_catalog ’material ’brass ’types ’drill ’diameters ’d_0.25)
```

The tool catalog structure could be simplified by deleting the nodes where a branch has only one sub-branch (for example, by not making “types” the only branch of “brass” and using the type names directly as the branches of “brass”). The extra nodes occur because the structure was built in anticipation of the need for multiple branches. The need never materialized, and the structure was not simplified, since the extra layer of structure is not computationally expensive but reprogramming is expensive.

Figure 5. Tool Catalog



IV. DYNAMIC DATABASES[1]

1. VWS WORLD MODEL

1.1. Overview

The property list of "vws_world" is used for information about the physical setup of the workstation and to keep information about data such as process plans, NC-programs, and machining parameters. The list is hierarchical. The only linking in the vws_world list is between the "machine_tool" and "tools" branches of the list, where there are pointers from both branches to tooling information.

The top level of the hierarchy includes 17 properties. Five of these (kit_orders, lots, orders, robot, and run_id) are not currently being used.

1.2. Tooling

Tooling is one of the few areas in which there is a need for data from the AMRF global database, but there is no AMRF database report specification for data. Experience in the VWS shows some of the data elements required for tools and should be helpful in pointing the way to a standard tool specification. However, the VWS tool representation has been applied to only a limited range of tool types, and it is clear that additional data about the types that are covered may be required for some applications. AMRF research associates from Texas Instruments are taking a deeper look at tooling than has been attempted in the VWS.

Two of the top-level properties (machine_tool and tools) in the vws_world property list are used primarily for tooling information. The machine_tool branch contains other information about the machine tool, but the other information is not in active use. The important part of the machine_tool branch is obtained by the command:

```
(fetch 'vws_world 'machine_tool 'tools_in_machine 'changer_list).
```

This is a branch in which the properties are changer-slot numbers, and the values are branches describing tools.

The same information about tools is kept in the "tools" branch of the vws_world where it is arranged by tool_id. The linking of tooling information is such that if, for example, the tool with tool_id 211 is in changer_slot 11, the following two commands will get the same data:

```
(fetch 'vws_world 'tools 'tool_descriptions 211)
(fetch 'vws_world 'machine_tool 'tools_in_machine 'changer_list 11)
```

Either of those two commands will retrieve the data shown in Table 4.

---

1. Dynamic databases are databases that change during normal VWS operation.

Table 4. Tool Data From VWS World Model

(211	id	211
	location	changer
	changer_slot	11
	type	end_mill
	tool_type_id	end_mill_0.125_2_ab
	cutting_depth	0.4
	exposed_length	0.625
	cutting_diameter	0.125
	shank_diameter	0.25
	tip	(tip tip_type flat)
	flutes	2
	materials	(aluminum brass)
	tool_material	steel
	type_of_use	general
	inches_cut	0)

As shown in Table 4, a tool has 15 properties. For taps there is a sixteenth property: threads_per_inch. Seven of the 15 (type, cutting_diameter, flutes, shank_diameter, cutting depth, tip, and materials) are the same as properties in the tool catalog for a tool of the given tool_type_id. The values of these seven should be the same as given in the tool catalog. The value of the "tool_type_id" property itself is the first atom of the entry in the tool catalog and is not carried as a property in the catalog.

Six of the properties of a tool in vws_world database are not in the tool catalog because they refer to a specific tool, not a tool_type_id.

A. id - the id of the specific tool.

B. location - where the tool is located. The value is either "spindle" or "changer". This data is not currently being updated or used.

C. changer_slot - the number of the changer slot the tool is in when it is not in the spindle.

D. exposed_length - the length of the tool sticking out of the tool holder. This data does not exist until the tool is put into a tool holder, so it cannot be put in the tool catalog.

E. type_of_use - whether the tool may be used for general purposes, finish milling, etc. This data is not currently being used.

F. inches_cut - the number of inches which the tool has cut in its lifetime. This data is not currently being updated or used.

The seventh property not in the tool_catalog, tool_material, is not currently being used. The tool catalog and the naming of tools probably should be changed to include this information.

1.3. Fixturing

Three branches of the vws_world property list are used for information about fixturing: vise, pallet, and fixture. The vise and pallet branches are changed only if the user makes changes. The fixture branch is changed automatically by the Data Execution module using data from the workpiece description and data from either the vise or pallet branch, as the user has directed.

Not all the data in the vise branch is used or kept up to date. The data which is used and kept up to date includes: the maximum and minimum opening of the vise jaws, the maximum part length, bottom obstacles, side obstacles, run_offs, bottom_clearance, and safe_z_plane. The way in which the last five items are used is described in [K&S2], Chapter II, section 2. There is no method of automatically updating these items if a fixtured part is loaded into the vise, however.

The pallet branch contains the same type of information as the vise branch, with length and width replacing maximum and minimum opening. Although fixturing information is automatically updated when the pallet is used, this includes no side obstacles. In order for the data to be adequate to support the automatic interference checking the system will perform, manual input by the user is required to describe side obstacles on the pallet.

1.4. Objects

The “objects” branch of the vws_world property list contains information about two types of objects that enter and leave the VWS in the course of operation: trays and workpieces. The type of data kept is specific to the type of object. The item_nr parameter is used as the property name for both types of object. This data changes when a tray enters or leaves the workstation. Workpieces enter and leave the workstation in trays.

1.5. Tray_defs

The “tray_defs” branch of the vws_world property list contains descriptions of types of trays. There may be several trays of a given type or none in the VWS at any given time.

1.6. Data About Data

Two branches of the vws_world property list contain data about data: process_plans and programs. The process plans are either workstation level plans or plans for the milling

machine. The programs are NC-programs for the milling machine. The actual plan or program is kept in the "database" directory discussed in Chapter II, section 3 of this paper. A set of information about the data, such as name, version number, length in bytes, etc., is kept in the property list. Plans and programs may be added or deleted on command from the user at the control system.

1.7. Other Data

Other small branches of the vws_world property list and their values are:

A. machining_parameters = (machining_parameters min_thick 0.06 max_thru 0.75)

B. supervisor = cell.

C. system id = vws.

2. LOCAL DATABASE MANAGER WORLD MODEL

The property list of "db_mgr_world" is used by the local database manager to help it emulate the AMRF database. It contains some of the information in the vws_world, plus information about process plans, NC-programs, trays, and workpieces that is not known to the vws_world. Because the local database manager was initially set up to serve several clients, the db_mgr_world has several branches not currently in use.

Several branches of db_mgr_world have the same format (but possibly different data) as the branches with the same names in vws_world. These include: programs, process_plans, and objects.

Branches of db_mgr_world not found in vws_world include:

A. mailbox_table - For each client system of the database manager (currently there is only one, the VWS control system) there are four common memory areas called mailboxes. Each mailbox is comprised of three data structures. This branch of db_mgr_world keeps the names of the three data structures for each of the four mailboxes. Thus, "mailbox_table" has one branch, "vws", which has four branches: "command", "status", "data_in", and "data_out", each of which has three branches: "name", "table", and "data".

B. log_line - a counter for lines of the "db_mgr_log" file.

C. report_names - a list of the names of the ten report types the database manager can deal with.

3. GRAPHICS DATA

When the graphics subsystem is working (for the Part Design Editor module, for the Data

Execution module, or when drawing a design or workpiece), it uses the property list of "drawp" to maintain global information. The following properties are included:

A. object - the name of the design or workpiece being drawn

B. verify_flag - the current verification setting (off, soft, or hard).

C. block_size - the length, width, and height of the block.

D. scale - the scale of the drawing in world coordinate units per inch. World coordinates are 1000.0 by 750.0, regardless of the size or location of the graphics window.

E. the x and y coordinates of four points - the coordinates (in the world coordinate system) of the lower left and upper right corners of the top view of the block and the lower left and upper right corners of the right side view of the block. On the top view the property names are xt1, yt1, xt2, and yt2. On the right side view the property names are xr1, yr1, xr2, and yr2.

F. mask_cmds - the LISP commands needed to draw the polygons that mask the drawing. The commands are stored whenever masking is required and executed whenever a "remask" command is given.

G. main_mask - the polygon command needed to make the mask which surrounds the original picture of the block.

H. last_seg_no - a counter that is incremented each time a new graphics segment is drawn.

I. segment_list - a list of the currently drawn segments

J. step_index - a branch in which the property names are the step numbers of steps from a process plan, and the value of a property is the name of the graphics segment which shows the feature or subfeature produced by that step.

K. feature_index - a branch in which the property names are the feature numbers of features in a design, and the value of a property is a list of the graphics segments which show the feature. As many as three segments may be needed to represent a feature.

L. nc_code - a list of NC-pseudocode produced by the Data Execution module. This property is used only when the Data Execution module is run with the NC-coding option on. The list is used for drawing a tool path.

M. feature_num - the number of the feature being drawn (used only in the Data Execution module).

4. DATA FOR NC-CODING

In order to maintain global information while it is at work, the NC-coding subsystem of the Data Execution module uses the property list of “mtool”. The list has four branches:

A. nc_code - a list of NC-pseudocode. The list is built in reverse order and added to by the “nc_line” and “nc_hunk” functions. The format of pseudocode is discussed in [KR&W] Chapter V, section 3.

B. tool - the changer slot number of the tool last used.

C. speed - the speed of the spindle used in the last machining operation.

D. file_name - the name of the file to which NC-code should be printed.

When NC-coding has been turned on in the Data Execution module, the mtool property list is set up by the “init_nc” function and set to nil by “close_nc”. When NC-coding is off, the list is set up as a dummy by “init_exec_plan” and set to nil at the end of module operation by “execute_plan”.

5. DATA EXECUTION MOCKUP

In order to maintain global information while it is at work, the Data Execution Module uses the property list of “mockup”. The list has three branches. The branches are formatted like a workpiece, a process plan, and a design. The formats are discussed elsewhere.

A. workpiece - a working copy of the model of the workpiece. This working copy is updated by the “execute_step” function when a step of a process plan is executed.

B. process_plan - the enhanced process plan being executed.

C. design - the enhanced version of the design specified in the process plan being executed.

The property list of “mockup” is set up by “init_exec_plan” during initialization of the module and wiped out by “execute_plan” at termination of the module.

6. PART DESIGNS

Part design data changes frequently in the VWS2 system. The function of the Part Design Editor module is to help create new design data and to help change existing data. The vws_cadm user interface may be used to load existing designs.

Part designs are given in LISP-readable form in unlinked hierarchical property lists. Part designs have an alternative, human-readable format, as well. The format of part design data is explained in great detail in [K&J2], Chapter II, section 2. Examples of part design data

structures are given in Tables 1, 2 and 3 of [K&J2], Table 1 of [KRA1] and Table 3 of [KR&W].

7. WORKPIECE MODELS

Workpiece model data also changes frequently. Workpiece models are created by a user on a text editor or by the vws-cadm user interface when the Data Execution module is used. Workpiece models are modified by the Data Execution module, which will save the models if the user so chooses.

The format of a workpiece model is an unlinked hierarchical property list very similar to a design. The format is discussed is [K&J2], Chapter II, section 2.12. An example is given in [KR&W], Table 1. Only LISP-readable forms are used.

8. PROCESS PLANS

Two types of process plans are used in the VWS: workstation-level process plans (operation sheets) which are executed by the VWS controller, and machine-tool process plans (instruction sets) which are executed by the Data Execution module.

Process plan data changes rapidly and is structured in LISP-readable format in unlinked hierarchical property lists. In addition to the LISP-readable forms which are used by the VWS2 system, there is an AMRF standard for process plan formats. The standard format is a hierarchical property list in concept, but it is not LISP-readable.

Details about both formats are given in [KRA1]. Examples are given in Tables 2, 3, and 4 of [KRA1] and in Table 2 of [KR&W].

V. LOCAL DATABASE MANAGER

1. OVERVIEW

The local database manager is a VWS2 data handling subsystem which acts as an easy-to-use interface between the VWS controller and either the AMRF global database or the local VWS database. In addition, the local database manager is able to emulate some functions of the AMRF database, so that workstation operations can be carried out in stand-alone data mode (stand-alone control mode is also possible, independently).

Figure 1 shows where the local database manager is located in the VWS2 system and how it is connected with other parts of the system.

The user of the VWS control system selects whether the local or AMRF global database should be used. The local option is available whenever the control system is running. The global option requires that the AMRF database be working, that the network between the AMRF shop and a remote VAX computer be working, and that Sun common memory be working. A switch to local data mode may be made any time the control system is ready to accept a command. A switch to global data mode may be made any time the control system is ready to accept a command if the three systems just mentioned are ready.

The local database manager cannot provide a full emulation of the global database because the VWS is not in contact with other workstations. Data such as the contents of a tray may be changed by other workstations. The global database normally will be kept informed of such changes so that it is able to provide correct information. The local database manager may be unaware of a change.

The AMRF global database system supports four types of transactions with workstations:

A. A workstation may send a command to the global database.

B. The global database will send one or more status messages to a workstation in response to a command, informing the workstation about the progress it is making in carrying out the command.

C. A workstation may send data in a report to the global database.

D. The global database will send data in a report to a workstation, if asked.

In order to conduct these transactions, communication between the workstation and the global database is required. This is provided by Sun common memory, using the AMRF network. In the VWS there are four common memory areas, called “mailboxes”, for conducting database transactions, one mailbox for each transaction type.

2. MODES OF OPERATION

The local database manager has three modes of operation. Within the local database manager they are called: pass, monitor, and local. The pass mode is used by the VWS controller for dealing with the AMRF global database. The local database manager local mode corresponds to the local data mode of the controller. The local database manager monitor mode is not currently used by the controller, although it is fully operational.

In pass mode the local database manager acts only as an interface between the controller and the global database.

In monitor mode the local database manager acts as an interface between the controller and the global database, but while it is doing so it monitors the exchanges between the controller and the global database. Specifically, it reads all commands, status messages, and reports passing between the VWS and the global database. It saves all information from reports in the db_mgr_world or the local database. The purpose of this is to keep the local database manager fully informed, so that if a switch to local data mode is made, it will be up to date.

In local mode the local database manager emulates the global database, receiving commands, sending status, receiving reports, and sending reports.

In order to change local database manager mode, a LISP command is given of the form:
(set_db_mgr_mode *'mode_name*)

where *mode_name* is one of: pass, monitor, or local.

The change of mode is accomplished when the “set_db_mgr_mode” command is given by redefining a set of five functions: db_read_report, db_write_report, db_read_status, db_write_command, and db_done_status.

The activities of the local database manager for the four database transactions in the three local database manager modes are summarized in Table 5.

Table 5. Local Database Manager Summary

TRANSACTION	MODE		
	PASS	***MONITOR***	***LOCAL***
1. Send database command.	Encode command, put it in command mailbox, and have communications system send it.	Encode command, put it in command mailbox, and send it. If command is insert, get report from outgoing mailbox and read it into db_mgr_world. If command is delete, delete from local list. Record transaction.	If command is select, write report from data in db_mgr_world; save in db_mgr_world. If command is insert, read saved report into db_mgr_world. If command is delete, delete from local list. Record transaction.
2. Read status.	Get message from status mailbox, decode it, and hand it to user.	Get message from status mailbox, decode it, and hand it to user. Record transaction.	Hand "done" status to user. Record transaction.
3. Read report	Get report from incoming data mailbox and read it into vws_world. If it is a process plan or control program report, write a file also.	Get report from incoming data mailbox and read it into vws_world and db_mgr_world. If it is a process plan or control program, write a file also. Record transaction.	Read report which is saved in db_mgr_world into vws_world. Record transaction.
4. Write report	Generate report from data in vws_world (and from a file if control program or process plan), and put report in outgoing data mailbox.	Generate report from data in vws_world (and from a file if control program or process plan), and put report in outgoing data mailbox. Record transaction.	Generate report from data in vws_world (and from a file if control program or process plan), and save it, unread, in db_mgr_world. Record transaction.

3. COMMON MEMORY INTERFACE

The interface between the VWS2 system and common memory was built by Dr. J. Jun and is not described in detail in this paper. See [LIBE] for a discussion of common memory.

When the interface with common memory is running and mailboxes are set up, each mailbox is represented in the VWS2 LISP environment by three structures which are of the LISP data type “vectori”. One represents the name of the mailbox, one is an area in which data (a “message”) is placed, and one is a table with information about the message, such as the number of bytes. The system on the other end of the communication line has three matching data structures.

The data area of a mailbox has a fixed length, but the messages put into it may vary in length. Bytes one and two of a message are a sequence number (a number which should increase by one each time a new message arrives in the mailbox). Bytes three and four give the number of bytes in the message - including the four bytes just mentioned. If the number of bytes in the message is N, then the information which is in the message is in bytes 4 through N of the data area of the mailbox. Of course, the number of bytes of information must not exceed the length of the mailbox minus 4.

To ship information from one place to another through a common memory mailbox, the sending system puts the sequence number, number of bytes, and information in the data area of its mailbox, updates the table and gives two common memory commands: cm_set_value and cm_sync. On the other end, the receiving system gives two commands: cm_sync, and cm_get_value. Then the data area of the receiving mailbox will have the message in it that was sent. These three common memory commands (cm_sync, cm_set_value, and cm_get_value) are the only common memory commands used by the local database manager (operating common memory communications requires additional common memory commands). Appropriate arguments to these functions are required.

The data in the data area of the receiving mailbox is not changed by the common memory system unless the receiver gives the cm_sync and cm_get_value commands. This fact is used by the local database manager to test whether the data is new.

4. DATABASE INTERFACE

4.1. Introduction

Specification of the interface between the local database manager and the AMRF global database has three parts:

A. allowable commands to the global database

B. status messages from the global database

C. reports sent to and from the global database.

All these specifications have been promulgated by the AMRF database project and are used throughout the AMRF. This paper gives only a brief description of them.

From the VWS point of view, the global database is passive. It does not issue commands or requests for data to the VWS. All data exchanges must be initiated by the VWS.

4.2. Commands

A database command is given to the global database by writing a Data Manipulation Language (DML) string, encoding it, placing it in the command mailbox, and sending it to the global database. Here are two examples of DML strings:

```
SELECT * FROM PROCESS_PLAN WHERE EXEC_SYSTEM = 'VWS' AND
PLAN_ID = 'FLAM' AND PLAN_VERSION = 1 USE MEMORY 'DS_VWS_DIN'

INSERT INTO PROCESS_PLAN VALUES USE MEMORY 'DS_VWS_DOUT'
```

The first DML string asks for a process plan from the global database. This is a SELECT command. It is typical of a SELECT command requesting a report from the global database. In requesting a report, the report is identified by giving the name of the type of report and the values of certain key fields in the report.

The second DML string sends a process plan to the global database, using an INSERT command. This is a typical DML string for sending a report. In sending a report, the only identification given is the type of report.

In both cases the name of the mailbox to use is given at the end of the string. The VWS incoming data mailbox is DS_VWS_DIN, and the outgoing data mailbox is DS_VWS_DOUT.

The two types of commands shown in the DML strings above (SELECT and INSERT) are the only types of DML strings currently used by the local database manager. The local database manager is set up for a third type of command, DELETE, which deletes a report from the global database, but the capability of executing a DELETE command has not been available in the global database. A fourth type of command, UPDATE, is available in the global database, but the local database manager has not needed to use it.

In local mode, the processing of commands is much simpler. Acting on behalf of the VWS controller, the local database manager creates a DML string. The same string is created regardless of which mode the manager is in. Then with its global database emulation hat on, the local database manager parses the DML string directly, without encoding it and without putting it in a mailbox (mailboxes are not even set up in local mode). The local database manager is capable of parsing any SELECT, INSERT, or DELETE command that it can write. After the command is parsed, it is carried out. If carrying out the command requires a report, the report is written and placed in the db_mgr_world until the control system asks to read it.

It would have been simpler, of course, to have the local database manager skip the writing and parsing of a DML string, but one of the uses of the local database manager was to serve in its emulation capacity to test its capabilities in its interface capacity. The software is in place for having commands encoded and decoded, as well, but it was decided to stop short of this extra bit of realism in emulation.

4.3. Status

When the global database is being used, status messages are sent to the VWS by the global database through the VWS database status mailbox. They are sent encoded and must be decoded after being retrieved from the mailbox. The status data sent by the global database includes several fields. Only two of these, “summary” and “detail”, are extracted by the local database manager, and only the “summary” is being used. The “summary” is an integer. Three values are defined:

B = (ASCII 66) = busy.

D = (ASCII 68) = done.

E = (ASCII 69) = error.

In local mode, a command is always executed before control of the LISP environment is returned from the local database manager to the VWS control system. If the command is not successfully executed, a message will be posted on the terminal, and LISP will halt. Thus, if the system has not halted, when control of the LISP environment is returned to the VWS control system, the command must have been executed successfully, i.e. the status must be “done”. Thus, in local mode, the LISP function which reports status is very simple: it always returns “done”.

4.4. Reports

Reports must be one of a fixed set of report types. For each type there is a specific format. Report formats are discussed in the next chapter. Messages which transmit reports either to or from the VWS are not encoded.

5. VWS CONTROLLER INTERFACE

5.1. Overview

The interface between the local database manager and the VWS controller was designed with two main objectives:

A. to make database transactions usable by simple LISP function calls in the controller.

B. to allow the controller to use the same function calls to deal with either the AMRF global database or the local VWS database.

As noted earlier, the local database manager runs in either the local mode or the pass mode when working for the VWS controller. Switching between the two may be done at any time by the VWS controller, with the limitations mentioned in the introduction to this chapter.

To make function calls simple, a class of LISP functions was written for each of three command types: “db_obtain” functions for SELECT, “db_insert” functions for INSERT, and “db_delete” functions for DELETE. When any of these functions is called, all the activities necessary to prepare for database transactions, carry out transactions, and follow up on them are performed by the local database manager. Specifics follow in subsections 5.2 through 5.4.

All the “db_obtain”, “db_insert”, and “db_delete” functions use the “db_done_status” function. In local mode, as already mentioned, db_done_status just returns “done”. In pass mode, however, it does a lot. First it keeps checking the VWS database status mailbox, waiting for a new status message to arrive, checking as often as possible whether the sequence number has changed. Each 100 times it checks without finding a new sequence number, it prints a message to the controller console. If a “busy” message is returned, db_done_status keeps checking and printing a message to the console every 100 tries. If a “done” message is received, a message to this effect is printed on the console, and “done” is returned. If an “error” message or any other type of status is received, “nil” is returned, and an appropriate message is printed to the console.

Two types of data, process plans and NC-programs, may be large and are not stored in the vws_world. Rather, they are kept in files in the database directory. The local database manager uses the database directory both for process plans and NC-programs that it knows in its AMRF global database emulation role and for process plans and NC-programs saved for the VWS. In order to keep track of which files are intended to be available in the emulation role, two lists are kept of the names of files: “process_plan_list” and “nc_list”. Each list is kept in a file in the database directory. If the VWS controller gives an INSERT command for either a process plan report or a control program report while the local database manager is in local mode, the name of the appropriate file is added to the appropriate list. Similarly, a DELETE command for those report types in local mode removes a name from a file. Neither command adds or deletes a process plan or control program file from the database directory.

A naming convention is used for process plan files in the database directory. The name of a file is the plan id, followed by an underscore, followed by the version number. Thus, the name of the file containing version 9 of the LOK process plan is LOK_9.

5.2. "db_obtain" Functions

Four "db_obtain" functions are available to the VWS controller. They are for NC-program, process plan, tray contents, and tray definition reports. These functions generate an appropriate DML SELECT string, send it as a command, wait for a "db_done_status" to return "done", and then call the appropriate report reader to transfer information from the report into the vws_world.

In the pass mode, process plans and NC-programs are taken out of reports received from the global database and written into files in the database directory. In local mode the files must already exist in the database directory in order for the report to be prepared, so no new files are written.

5.3. "db_delete" Functions

Two "db_delete" functions are available. They are for NC-program and process plan reports. These functions generate an appropriate DML DELETE string, send it as a command, and wait for a "db_done_status" to return "done".

5.4. "db_insert" Functions

Two "db_insert" functions are available. They are for NC-program and process plan reports. These functions start by updating the vws_world from the appropriate file in the database directory. Then they call a report writer to write a report. Next an appropriate DML INSERT string is generated and sent as a command. Finally, they wait for "db_done_status" to return "done".

VI. REPORTS SYSTEM

1. OVERVIEW

As discussed earlier, information is passed between AMRF workstations and the AMRF global database via reports. A report is a string of bytes to be interpreted as ASCII characters. The formats for 33 workstation report types are given in [PO&M].

The VWS2 local database manager can deal with ten report types on behalf of the VWS control system. In its capacity as an emulator of the AMRF global database it can deal with three of these. It would be a simple matter to add functions for reading and writing the other seven report types to the emulation capability. Arranging to get useful data to put into the reports might be harder. The database manager's world model would have to be expanded to hold new types of data, and functions would have to be written to extract it.

The three report types for which emulation is in place (control_program, process_plan, and tray_contents) are the only three currently being used by the VWS control system.

The essence of writing a report is to extract information from local data structures and generate a string of bytes in the proper report format. Reading a report is the reverse process: extracting information from a string of bytes and putting it into local data structures (and, possibly, adding accessing information to a local data structure - updating an index, for example).

Some reports types are intended to be written only by a workstation, some to be read only, and some to be both written and read. Nevertheless, functions to both read and write are in place for every report type supported by the local database manager.

The functions which read and write reports are generated automatically, as discussed in the next chapter. The average length of one of them is about a page. Two examples are given in the next chapter.

The local database manager has the capability to print a nicely formatted version of any of the ten report types supported. This is intended to make it convenient for a human to read a report. The VWS is automated to such an extent, however, that no use is currently being made of this "pretty-printing" capability.

2. REPORTS SUPPORTED

The following ten report types are supported:

A. control_program - Read and writing a control_program report involves reading or writing a file as well as extracting or inserting information from a data structure in active memory.

B. equip_item_action

C. item_status

D. kit_order

E. order_action

F. order_performance

G. order_status

H. process_plan - Read and writing a process_plan report involves reading or writing a file as well as extracting or inserting information from a data structure in active memory.

I. tray_contents

J. tray_definition

VII. AUTOMATIC GENERATION OF REPORT READERS AND WRITERS

1. OVERVIEW

AMRF global database reports must be readable and writable by the VWS2. Any report the global database can send to the VWS must be writable by the local database manager when it is emulating the global database and readable by the local database manager both for its own use and for use by the client system. Any report the VWS can send to the global database must be writable by the local database manager acting on behalf of the client, and readable by the local database manager acting on its own behalf or emulating the global database.

In addition, it is desirable to be able to print any report in a format easy for a human to read.

The report types which are defined for workstations in the AMRF are given in [PO&M]. That paper prescribes the format for reports and the meaning of their contents. Most formats have been somewhat changed by AMRF database project personnel since the publication of [PO&M]. The changes are not documented. All known changes are incorporated in the VWS reports system.

The VWS2 system includes a subsystem which automatically generates the LISP functions that read and write reports. We will call this subsystem the "generator" in this chapter. For each report type, the generator requires a table prepared by the systems programmer (the author in all cases so far). The generator includes a function that will print a nicely formatted copy of a report of any type for which a table has been prepared.

2. TABLE STRUCTURE

The table used by the generator to handle a report type has five columns and may have any number of rows. The table used for the "tray_contents" report type is shown in Table 6. (Table 6 is out of date. Some recent changes in format are not included in the table.)

The first column, "Field Name", contains the name of the data item that the line deals with. Names are as given in [PO&M]. The term "nil" is also allowable in the field name column. The meaning of "nil" is explained below.

The second column gives the type of data in the field. Allowable types are:

A. char - character

B. int - integer

C. int_a - array of integers, each assumed to be eight bytes long

D. nil - used whenever the field name is nil.

The third column gives the width of the field in number of bytes.

The fourth column, “Read Action”, contains either a LISP atom which serves as a placeholder or a LISP list which indicates what action should be taken by the report reading function when this field is encountered. The fourth column may be left blank in some cases.

The fifth column, “Write Action”, contains either a LISP atom which serves as a placeholder or a LISP list which indicates what action should be taken by the report writing function when this field is encountered. The fifth column may be left blank in some cases, for example on the line in Table 6 with nil in the field name column.

Table 6. Generator Table for Tray Contents Report

TRAY_CONTENTS REPORT FORMAT

Field Name	Type	Width	Read Action	Write Action
report_name	char	32	(db_test (eq val ’tray_contents))	’tray_contents
container_nr	char	16	(db_activate ’objects)	(get (db_active) ’item_nr)
nil	nil	0	(db_do (place (db_active) ’item_nr (car (db_active))))	
item_name	char	16	(place (db_active) ’item_name)	(get (db_active) ’item_name)
tray_clear	int	16	(place (db_active) ’tray_clear)	(get (db_active) ’tray_clear)
nr_locations	int	16	(db_put_tray_objects (db_active))	(db_get_tray_objects (db_active))
sector	char	16	sector	sector This is a comment
item_nr	char	16	item_nr	item_nr
item_name	char	16	item_name	item_name

end

In this report it will be assumed that if a tray location is empty, it will not be included in nr_locations.
Thus if a tray has four sectors and three of them are full, nr_locations will be 3.
This avoids having to consult the tray definition when writing the report.

3. TABLE READER AND COMMENTS

A table reader reads the table into the LISP environment for the generator. The first thing the table reader looks for is a row of at least 8 asterisks. Thus, the table heading is ignored by the system. It is included for human reading.

After finding a row of asterisks, the reader reads line-by-line and saves each line as a list, until it comes to a blank line. Then it stops reading. Anything below the blank line is ignored by the reader. As a convention for humans, all tables have the word “end” on a line following the blank line. All the lines following the blank line are commentary for the benefit of human users. Three lines of commentary are shown in Table 6. The final output of the reader is a list of lists, each of which was a line in the table.

The generator only pays attention to the first five entries on each line, so comments may be included at the end of a line without any special delimiters, as long as they are LISP-readable. One comment, “This is a comment”, is shown on the line whose field name is “sector”. Although no comment delimiter is required, comments should not appear at the end of lines with fewer than five entries, or they will be treated as data.

4. CAPABILITIES AND TERMS

4.1. Use of “nil”

The generator constructs a segment of LISP code for each line in the table when it is generating a report reading or writing function. Each segment of code reads or writes some of a report, except when the field name entry is “nil”. When the field name is “nil”, the generator inserts a line of LISP code in the function being written, as specified in the read_action or write_action column. Thus “nil” serves as a placeholder in the field name column when it is desired to insert some extra LISP code in a report reader or writer.

If the field name is “nil”, then the type should also be “nil”, and the entry in the “width” column should be zero.

4.2. Use of “db_test”

If a read action column entry is a list starting with “db_test”, that indicates that the rest of the entry is a LISP form which should not evaluate to nil. The intent is that the rest of the entry be a should be a test involving the data in the field. A local variable named “val” is used in every report reader. The value of “val” is set to the data in the field being tested whenever a “db_test” is encountered. Thus, the test normally involves the local variable “val”.

One “db_test” occurs in Table 6. The test is that the report name should be “tray_contents”.

4.3. Use of “db_do”

If a read or write column action entry is a list starting with “db_do”, the generator uses the LISP form which is the second entry in the list as the segment of code to be inserted at that point in the function.

4.4. Use of “db_activate” and “db_active”

In order to transfer data between database reports and the vws_world or the db_mgr_world, it is useful to have a hook into a branch of the world. This is provided by the “db_activate” and “db_mgr_activate” functions. Executing “db_activate” redefines the function “db_active” so that the function call (db_active) returns a pointer to a branch of the vws_world. Executing “db_mgr_activate” does the same for the db_mgr_world. Both functions build branches if they do not already exist.

For example, the function call (db_activate 'objects) causes the function call (db_active) to return the same thing as the function call (fetch 'vws_world 'objects).

Once a branch of a world model has been activated by a call to "db_activate" or "db_mgr_activate", data may be inserted in the branch by a function call of the form (place (db_active) ...). Data may be extracted from the branch by a call of the form (fetch (db_active) ...). There are several example of both types of calls in Table 6.

4.5. Use of "nr_" for Multiple Instances

If a report allows a variable number of instances of data of a certain type, this is flagged by using "nr_" (an abbreviation of "number_of_") as a prefix to the field name. In Table 6, for example, there is nr_locations. Following any such term in the field column of the table are the names of one or more subfields. In Table 6 there are three subfields: sector, item_nr, and item_name. Subfield names are distinguishable because the entry in the read and write action columns for a subfield will either be blank or be the same as the subfield name.

There is one special case of subfield names: if there are two subfields and their names are "attribute" and "value". This indicates, obviously, that the subfields are attribute-value pairs.

To deal with a multiple number of instances, a data structure convention has been adopted as follows. The data structure corresponding to the multiple instances will be a property list. The first entry in the list is always the parent field name (without the "nr_" prefix). If the subfields are attribute-value pairs, the rest of the entries are arranged attribute, value, attribute, value, etc. Otherwise each set of subfield values is arranged in a property list with the subfield names as the properties, and the lists are assembled into a property list in which the property names are the positive integers. Suppose, for example, that in a tray contents report the bytes starting at nr_locations were as follows (where a period is used to indicate a blank):

2.....top_half.....3459812.....gizmo.....bottom_half.....419007.....whatsis.....

The corresponding data structure would look like:

```
(locations 1 (1  sector  top_half  item_nr  3459812  item_name  gizmo)
             2 (2  sector  bottom_half  item_nr  419007  item_name  whatsis))
```

The report reader function must build the list from the row of bytes, and the report writer function must build the row of bytes from the list. In some cases the list may be extracted from or inserted in the world model without change. In other cases special extraction or insertion functions are required.

4.6. Variable Length Fields

The generator can deal with variable length fields (where the length of the field is specified in the report text) as long as there is only one such field in a report and it is the last field in the report. There are two examples in the current system: process plan and control program reports.

5. HOW THE READER GENERATOR WORKS

5.1. Overview

To generate a LISP function to read reports of a given type, the generator does the following:

A. Call the table reader to read the table, creating a format list.

B. Go through the format list and generate all the tests indicated by “db_test” (see part B of Table 7).

C. Go through the format list again and generate the body of the function (see subsection 5.2 and part C of Table 7).

D. Cull out or consolidate “setq n” lines (see subsection 5.3).

E. Put three lines of LISP code (see subsection 5.4 and part A of Table 7) ahead of the tests.

F. Put one line of code (part D of Table 7) at the end to close the function.

G. Execute the assembled code to define the function, and print the function to the proper file.

Table 7. Tray Contents Report Reader

```
(def tray_contents_read
   (lambda (report)                                                    A
      (prog (n val)
            (setq n 0)
            (setq val (vreadchar report n 32))
            (cond                                                      B
             ((null (eq val ' tray_contents)) (return nil)))
            (setq n 32)
            (db_activate ' objects (vreadchar report n 16))
            (setq n (plus n 16))
            (place (db_active) ' item_nr (car (db_active)))
            (place (db_active) ' item_name (vreadchar report n 16))
            (setq n (plus n 16))
            (place (db_active) ' tray_clear (vreadchar report n 16))
            (setq n (plus n 16))
            (setq val (vreadchar report n 16))
            (setq n (plus n 16))
            (do* ((m 1 (add1 m))
                  (stop (add1 val))
                  (liz (list ' locations) (cons (reverse subliz) liz))
                  (subliz (list m) (list m)))
                 ((equal m stop)
                  (db_put_tray_objects (db_active) (reverse liz)))
                 (push m liz)
                 (push ' sector subliz)
                 (push (vreadchar report n 16) subliz)
                 (setq n (plus n 16))
                 (push ' item_nr subliz)
                 (push (vreadchar report n 16) subliz)
                 (setq n (plus n 16))
                 (push ' item_name subliz)                             C
                 (push (vreadchar report n 16) subliz)
                 (setq n (plus n 16)))
            (return t))))                                              D
```

5.2. Generating the Body of the Function

As noted earlier, a segment of code is written for each line in the table. The nature of the segment varies according to the line. Some actions were described earlier. If the entry in the read action column is not “nil”, a “db_test”, a “db_do”, or a placeholder for a subfield, the reader generator uses the entry as the code segment with one modification. The entry is always list. The modification is that a LISP form is inserted as the last element of the list. The LISP form gets data out of the report, structures it, and returns the structure.

In the case of a field name with the “nr_” prefix, an entire “do” loop is inserted in the body of the function. The subfields are embedded in the body of the “do” loop.

5.3. Culling and Consolidating Counter Resettings

The report reader which is being generated will keep track of where it is in reading a report by incrementing a counter with the classical name “n”. Both the test section and the body of the reader are initially written with many lines of the form (setq n (plus n ...)). In some places, such as the next_to_last line of the function, such lines may be completely discarded. If two or more consecutive lines of this type appear, it may be possible to consolidate them into one. For example, the two consecutive lines (setq n (plus n 3)) and (setq n (plus n 4)) may be consolidated into (setq n (plus n 7)). The generator does this culling and consolidating.

5.4. First Three Lines

The first three lines of a report reader always have the form:

```
(def reader_name
   (lambda (report)
     (prog (n val)
```

The first line indicates that a function is to be defined of the name reader_name.
The second line indicates that it is a lambda function and has one argument, which should evaluate to the report to be read. The third line indicates that the rest of the function is a “prog” with two local variables: n (the line counter) and val (a handy local variable used for several purposes).

6. HOW THE WRITER GENERATOR WORKS

6.1. Overview

To generate a LISP function to write reports of a given type, the generator does the following:

A. Call the table reader to read the table, creating a format list.

B. Go through the format list and prepare:
   a. a “size_list” of segments of LISP code needed to calculate how long the report must be.
   b. a “let_list” of local variables needed for the report and LISP code segments for calculating their value. These will be used to construct a segment of LISP code that will go just ahead of the body of the function. The segment will have three lines if the “let_list” is not empty, one if it is empty (see part B of Table 8).

C. Go through the format list again and generate the body of the function (see subsection 6.2 and part C of Table 8).

D. Cull out or consolidate “setq n” lines from the body of the report (see subsection 5.3).

E. Put two lines of LISP code (see part A of Table 8) at the beginning of the function to give its name and function discipline.

F. Put one line of code (part D of Table 8) at the end to close the function by returning the report.

G. Execute the assembled code to define the function, and then print the function to the proper file.

Table 8. Tray Contents Report Writer

```
(def tray_contents_write                                              A
   (lambda nil
      (let ((locations (db_get_tray_objects (db_active))) (n 0) size report)
         (setq size (plus 96 (times (quotient (length locations) 2) 48)))   B
         (setq report (new-vectori-byte size 32))
         (vsetchar report n 'tray_contents)
         (setq n (plus n 32))
         (vsetchar report n (get (db_active) 'item_nr))
         (setq n (plus n 16))
         (vsetchar report n (get (db_active) 'item_name))
         (setq n (plus n 16))
         (vsetchar report n (get (db_active) 'tray_clear))
         (setq n (plus n 16))
         (vsetchar report n (quotient (length locations) 2))
         (setq n (plus n 16))
         (do* ((liz (cddr locations) (cddr liz))
               (subliz (car liz) (car liz)))
              ((null liz))
              (vsetchar report n (get subliz 'sector))
              (setq n (plus n 16))
              (vsetchar report n (get subliz 'item_nr))
              (setq n (plus n 16))                                     C
              (vsetchar report n (get subliz 'item_name))
              (setq n (plus n 16)))
         report)))                                                     D
```

6.2. Generating the Body of the Function

As noted earlier, a segment of code is written for each line in the table. The nature of the segment varies according to the line. Some actions were described earlier. If the entry in the write action column is not “nil”, a “db_do”, or a placeholder for a subfield, the writer generator uses the entry without modification as the code segment.

In the case of a field name with the “nr_” prefix, an entire “do” loop is inserted in the body of the function. The subfields are embedded in the body of the “do” loop.

7. HOW THE TABLE IS USED FOR PRINTING REPORTS

A single function, “pp_report”, is used to print any type of report in a human-readable format. Like the function generators, it starts by calling the table reader to create a format list. It then goes through the format list and the report simultaneously, reading the report and printing out fields from the report according to the data type of the field. Special action is required if a field name has the “nr_” prefix. The report printer does not print out the text of the variable-length fields named “plan_text” and “prog_text” which occur in process plan and control program report types, respectively. For these two field names “Text follows.” is printed.

An example of a pretty-printed tray_contents report is shown in Table 9. The report was printed by the “pp_report” function using the table shown in Table 6. The report was prepared by the VWS2 local database manager.

Table 9. Pretty-Printed Tray Contents Report

```
                    REPORT
          ----------------------------------

            report_name       = tray_contents
            container_nr      = TRAY_200
            item_name         = |3_sector_tray|
            tray_clear        = 100
            nr_locations      = 3
                sector_1      = SECTOR_3
                item_nr_1     = PN111
                item_name_1   = LCLEVIS_BV

                sector_2      = SECTOR_2
                item_nr_2     = PN110
                item_name_2   = LCLEVIS_BV

                sector_3      = SECTOR_1
                item_nr_3     = PN109
                item_name_3   = LCLEVIS_BV

          ----------------------------------
```

8. HOW TO ADD A REPORT TYPE TO THE SYSTEM

8.1. Create the Table

To add a report type to the system, according to a format specified by the AMRF database project, the first step is to make a table for the generator corresponding to the given format. This has been done by the author for 10 report types using a text editor.

The first three columns of the table are easy to fill in, because all the data may be taken directly from the format document. The read and write action columns are more difficult to write. Preparing these columns requires intimate knowledge of LISP, of the capabilities of the generator, and of the structure of the vws_world database.

In the read action column, the lines on which the value in the report must be the same as some known value are identified. A “db_test” statement is written for each such line. For the other lines it is necessary to determine the structure of the relevant branch of the vws_world and to write entries that will put the data extracted from the report into the correct places in the structure. If the structure is not defined, it must be invented. If the existing structure makes the insertion of data awkward, special functions may have to be written for inserting data. In Table 6 an example of such a special function is “db_put_tray_objects”.

In the write action column, if a known value must be inserted in the report, the known value is used as the column entry. For the purposes of report writing, it is assumed that the user or the VWS control system will have made a call to “db_activate” before the report is to be written. Thus, the other lines of the write action column need only specify how to get a particular data item or substructure out of the data structure that (db_active) points to.

In some cases, the order of the fields in a report may be different from the hierarchical order of the corresponding branches of the vws_world. These can often be dealt with by the use of “db_do”. A “db_do” line may also be needed if two or more actions are to be taken on one piece of data. The “db_do” line in Table 6 resulted from the second of these two situations.

Some report types need only be read by the VWS and some need only be written. For a report that is not to be writable, the write action column may be left blank. For a report that is not to be readable, the read action column may be filled up with legitimate LISP forms of any sort; a good choice is “nil”.

8.2. Call the Generator

Once the table for a report type is complete, the rest is trivial. To make a report reading LISP function, make a function call of the following sort:

(gen_report_reader *input_file reader_name output_file*)

In the function call,

input_file = the UNIX name of the file in which the table is to be found.
reader_name = the name of the function to be written.
output_file = the UNIX name of the file in which the function is to be printed.

When this function call is completed, the reader function will be defined in the LISP environment as well as being printed to the specified file. The file names may include the usually UNIX indications for parent directories and subdirectories.

The report writer is generated by a function call of the form:

(gen_report_writer *input_file writer_name output_file*).

Once the functions are generated, they may be treated like any other LISP function: compiled, edited, etc.

8.3. Repeat for Local Database Manager

In order to get the local database manager to emulate the AMRF global database with respect to a new report type, this whole process must be repeated. Usually this will be relatively easy, since the table for the VWS may be largely copied to make the new table, and the same kinds of data structures may be used. Of course, in some cases, the VWS will need a reader or a writer only while the local database manager needs the opposite or both.

9. OBSERVATIONS REGARDING THE GENERATOR

The generator has proved to be an extremely useful subsystem. It writes LISP code for reading and writing database reports that is as efficient and concisely written as if the author had written the code directly. It writes the code correctly on the first try, which the author almost never does, and it writes it many times as fast as is humanly possible. The function in Table 7 was generated in 15 seconds by the system.

If a change is made in the format of a report, the report readers and writers may be updated quickly and correctly by editing the table for the report type and regenerating the reader and the writer.

It would be possible to regenerate and execute a report reading or writing function each time a report was to be written or read, but this would be a great waste of time. Report readers and writers change relatively infrequently, so it is better to save them and reuse them.

The generator's main drawback is the time and knowledge it takes to construct the read action and write action columns of a table and prepare the special functions and data structures required by a table.

Although the generator has no known bugs, it is sensitive to errors in the input table. It performs a few checks on input data, but is relatively easily thrown into an error state by bad input data. It would be a straightforward matter to make the generator more robust by writing routines to check the data in the input table.

The generation system is a good example of the type of application to which LISP is well-suited. It is typical of LISP to allow construction of a relatively sophisticated system which manipulates lists, using a small number of fairly brief functions, most of which are hard to understand.

The generator described in this chapter is one of two LISP function generating subsystems in the VWS2 system. The other generates functions which verify features in part designs. It is described in [K&S2], Chapter XIII.

It would be feasible in a matter of weeks to write similar systems in LISP to generate report readers and writers in other computer languages.

VIII. SOFTWARE

1. FILE STRUCTURE

1.1. Introduction

Most of the VWS2 software written by the author for data handling is kept in the directory ~kramer/vws2/datab2. That directory has six subdirectories, as follows.

1.2. Subdirectory “asn_c”

The “asn_c” subdirectory contains routines written in the “C” programming language for encoding and decoding database commands. None of these routines was written by the author. They were provided to the author by Don Libes. These routines are used by the local database manager for sending commands to the AMRF global database. It includes 13 uncompiled files (.h or .c suffix) and four complied files (.o suffix), as well as a Makefile.

1.3. Subdirectory “asn_lisp”

The “asn_lisp” subdirectory contains three LISP functions written by the author for interfacing the “C” routines of the asn_c directory with the rest of the VWS2 system.

1.4. Subdirectory “db_mgr”

The “db_mgr” subdirectory contains the software for the local database manager. It includes about 67 files. Of these, 9 are for reading and writing the three report types supported by the local database manager, 4 are for parsing DML strings, 8 are “db_obtain”, “db_delete”, or “db_insert” functions (described in Chapter V, section 5), 3 are solely for testing, and the rest are split between enabling the local database manager to emulate the AMRF global database and enabling the local database manager to serve as an interface between the VWS control system and the AMRF global database.

1.5. Subdirectory “reports”

The “reports” subdirectory contains 15 functions written by the author for the reading, writing, and printing of database reports. It also contains 40 other files, four for each of ten report types. The first portion of the name of each of these files is the report name. The ten report names are: control_program, equip_item_action, item_status, kit_order, order_action, order_perf, order_status, process_plan, tray_contents, and tray _def.

The four file types are:

A. A form to be used by the report generation subsystem. The format, construction, and use of these form files was described in Chapter VII. The name of each form file is the report name with the suffix “.form”. All the “.form” files were written by the author.

B. A pretty-printed report prepared by the system from a report written by the report-writing function for that report type, using test data. The name of each report file is the report name with the suffix ".rpt". These files have no active function in the system.

C. A LISP function to read a report of that report type. The name of each report-reading file is the report name with the suffix "_read.l". The name of the function is the same as the file name, without the terminal ".l". These files were all written by the report generation system.

D. A LISP function to write a report of that report type. The name of each report-reading file is the report name with the suffix "_write.l". The name of the function is the same as the file name, without the terminal ".l". These files were all written by the report generation system.

1.6. Subdirectory "rptgen"

The "rptgen" subdirectory contains 15 LISP functions for the automatic generation of report-reading and report-writing functions. These functions are heavily larded with backquote macros and are quite difficult to read. They would be totally unintelligible without the backquote macros. They average about 20 lines in length.

1.7. Subdirectory "world"

The "world" subdirectory contains several LISP-readable files which are used set up hierarchical property lists. Some of the files set up lists when they are loaded into a LISP environment. Others define functions when they are loaded, and the functions must be executed in order set up lists. In general, if it is expected that a list will need to be initialized only once, the file sets it up directly, and if it is expected that a list will be reinitialized from time to time, it will be set up by executing a function.

There are eight files of this sort in the world subdirectory.

A. "char_desc.l" sets up the plain font and the ASCII table needed for translating strings into character lists. The other four fonts currently used in the VWS2 system are built after this file is loaded by executing the "make_fonts" function.

B. "features.l" sets up the features database when it is loaded.

C. "verify_data.l" adds information to the features database when it is loaded. This data is used only by the automatic feature verification function writing subsystem. An example of this data is shown in the left-hand column of Table 2 in this paper. The use and format of the data is described in chapter XIII of [K&S2].

D. “init_vws_world.l” defines the function “init_vws_world” when it is loaded. Executing the function sets up the “vws_world” world model in skeletal form. The model which is set up contains no tooling information or data about any process plans, NC-code, trays, etc.

E. “init_tools.l” defines the function “init_tools”. The function should be executed after the vws_world has been set up by init_vws_world. When it is executed, init_tools sets up all data about tools needed in the vws_world database. It puts links in the database so that the same information about a tool is available by either of two routes. Specifically, if “tool_id” is a LISP variable which evaluates to the tool_id of the tool in the changer slot which is the value of the LISP variable “changer_slot”, then the following two function calls will access the same data substructure:

```
(fetch 'vws_world 'tools 'tool_descriptions tool_id)
(fetch 'vws_world 'machine_tool 'tools_in_machine 'changer_list changer_slot)
```

F. “init_vws_part_data.l” defines the function “init_vws_part_data”. When it is executed, this function initializes data about five parts which is required for making these parts in the VWS. The data is incorporated in the vws_world data structure.

G. “machine_ops.l” sets up the machine_ops database when it is loaded.

H. “tool_catalog.l” sets up and links the tool_catalog data structure when it is loaded. The linking is such that (for example) the following two function calls return the same data substructure:

```
(fetch 'tool_catalog 'names 'tap_0.194_4_abs)
(fetch 'tool_catalog 'material 'aluminum 'types 'tap 'threads_per_inch 24
    'diameters 'd_0.194)
```

In addition to the eight files just described, the world subdirectory contains eight functions used for making new fonts, and several miscellaneous functions, most of which access or alter the vws_world data structure. The font-making functions are described in chapter II, section 2.16 of [K&J2].

REFERENCES

[BAR&]
Barkmeyer, E. J.; Mitchell, M. J.; Mikkilineni, K.P.; Su, S. Y. U.; and Lam, H.; "An Architecture for Distributed Data Management in Computer Integrated Manufacturing"; NBSIR 86-3312; January 12, 1986.

[FUR&]
Furlani, C. M.; Libes, D.; Barkmeyer, E.; Mitchell, M.; "Distributed Databases on the Factory Floor"; Proceedings of the First International Symposium on Computerization and Networking of Materials Property Databases; November 1987; Philadelphia, Pennsylvania; ASTM; 1987.

[JUN]
Jun, Jau-Shi; "The Vertical Machining Workstation Systems"; NISTIR 88-3890; National Institute of Standards and Technology; 1988; 65 pages.

[KRA1]
Kramer, Thomas R.; "Process Plan Expression, Generation, and Enhancement for the Vertical Workstation Milling Machine in the Automated Manufacturing Research Facility at the National Bureau of Standards"; NBSIR 87-3678; National Bureau of Standards; 1987; 56 pages.

[KRA2]
Kramer, Thomas R.; "Process Planning for a Milling Machine from a Feature-Based Design"; Proceedings of Manufacturing International Meeting; April, 1988; Atlanta, Georgia; ASME; Vol. III, pp. 179 - 189.

[KRA3]
Kramer, Thomas R.; "The Graphics Subsystem of the Vertical Workstation of the Automated Manufacturing Research Facility at the National Bureau of Standards"; NBSIR 88-3783; National Bureau of Standards; 1988; 27 pages.

[KRA4]
Kramer, Thomas R.; "The vws_cadm User Interface in the Vertical Workstation of the Automated Manufacturing Research Facility at the National Bureau of Standards"; NBSIR 88-3738; National Bureau of Standards; 1988; 110 pages.

[K&J1]
Kramer, Thomas R.; and Jun, Jau-Shi; "Software for an Automated Machining Workstation"; Proceedings of the 1986 International Machine Tool Technical Conference; September 1986; Chicago, Illinois; National Machine Tool Builders Association; 1986; pp. 12-9 through 12-44.

[K&J2]
Kramer, Thomas R.; and Jun, Jau-Shi; “The Design Protocol, Part Design Editor, and Geometry Library of the Vertical Workstation of the Automated Manufacturing Research Facility at the National Bureau of Standards”; NBSIR 88-3717; National Bureau of Standards; 1988; 101 pages.

[K&S1]
Kramer, Thomas R.; and Strayer, W. Timothy; “Error Prevention in Data Preparation for a Numerically Controlled Milling Machine”; Proceedings of 1987 ASME Annual Meeting; ASME; 1987; PED-Vol. 25, pp. 195 - 213.

[K&S2]
Kramer, Thomas R.; and Strayer, W. Timothy; “Error Prevention and Detection in Data Preparation for the Vertical Workstation Milling Machine in the Automated Manufacturing Research Facility at the National Bureau of Standards”; NBSIR 87-3677; National Bureau of Standards; 1987; 61 pages.

[KR&W]
Kramer, Thomas R.; and Weaver, Rebecca E.; “The Data Execution Module of the Vertical Workstation of the Automated Manufacturing Research Facility at the National Bureau of Standards”; NBSIR 88-3704; National Bureau of Standards; 1988; 58 pages.

[KRI&]
Krishnamurthy, V.; Su, S. Y. W.; Lam, H.; Mitchell, M. J.; Barkmeyer, E. J.; “A Distributed Database Architecture for an Integrated Manufacturing Facility”; Proceedings of Conference on Data and Knowledge Systems for Engineering and Manufacturing; October 1987; Hartford, Connecticut.

[LIBE]
Libes, Don E.; “User-Level Variables (in a Hierarchical Control Environment)”; Proceedings of 1985 USENIX Conference; Portland, Oregon; 1985.

[LI&B]
Libes, D.; and Barkmeyer, E. J.; “The Integrated Manufacturing Data Administration System (IMDAS) - An Overview”; International Journal of Computer Integrated Manufacturing; Vol. 1, NO. 1; pp. 44 - 49.

[LOVE]
Lovett, Denver; “The Vertical Workstation’s Equipment Controllers”; NBSIR 88-3769; National Bureau of Standards; 1988; 59 pages.

[NA&J]
Nakpalohpo, Ibrahim; and Jun, Jau-Shi; “Automated Equipment Program Generator and Execution System of the AMRF Vertical Workstation”; not yet published; 1987; 17 pages.

[PO&M]
Potts, Michelle; and McLean, Charles; “Control-Database Interface: Workstation Level Reports”; AMRF internal report; October 9, 1986.

[RUDD]
Rudder, Frederick; “Operations Manual for the Automatic Operation of the Vertical Workstation”; NISTIR 89-4031; National Institute of Standards and Technology; 1989; 33 pages.

[RYB&]
Rybczynski, Siegfried; Barkmeyer, Edward J.; Wallace, Evan K., Strawbridge, Michael L.; Libes, Don E.; and Young, Carol V; “AMRF Network Communications”; to be published as an NBSIR; 1988.