R. H. Taylor
P. D. Summers
J. M. Meyer
IBM T. J. Watson Research Center
Yorktown Heights, New York 10598

AML: A Manufacturing Language

Abstract

AML, A Maufacturing Language, was designed to be a well-structured, semantically powerful interactive language for robot programming. In this paper, we identify the design objectives for such a language and give a technical description of AML. Important features are described and illustrated through representative examples of robot applications programming.

1. Introduction

One of the most significant developments in the 1980s will be the widespread introduction into industry of a new generation of intelligent automation systems, in which industrial robots operate under programmed control of modern digital computers. These systems combine several fundamental capabilities:

Manipulation—the ability to move physical objects about the work station
Sensing—the ability to obtain information from the physical world through the use of sensors
Intelligence—the ability to use information to modify system behavior in preprogrammed ways
Data processing—the ability to interact with a data base to keep records, generate reports, and to control activity at the work station

The flexibility inherent in this combination offers important advantages over current fixed automation. Positional misalignments can be determined by sensing and compensated for by software, thus reducing the requirement for expensive fixtures and high absolute accuracy in the robot hardware. Inspection can be integrated into automated manufacturing processes. Manufacturing data bases can be used to permit customization of individual work pieces. Similarly, model changes can be accommodated by software, thus permitting capital costs to be spread over many products. Finally, increased standardization of equipment within a plant eases maintenance and allows more flexible production scheduling.

If these advantages are to be readily obtained, however, it must be easy for a user to specify how the functional capabilities of the automation system are to be applied to particular tasks. Consequently, the provision of a suitable programming language and environment is a critical element for any successful system design.

The central role of programmability has long been recognized in university research laboratories (Taylor 1976; Lozano-Perez and Winston 1977; Grossman and Taylor 1978; Popplestone, Ambler, and Bellos 1978; Mujtaba and Goldman 1979; Park and Burnett 1979) and has recently received attention in industry as well (Olivetti n.d.; Unimation 1979; Will and Grossman 1980; VanderBrug 1981). For the past 10 years, the Automation Research Project at the IBM research facility in Yorktown Heights, New York, has been developing programmable automation systems (Will and Grossman 1975; Blasgen and Darringer 1977; Evans et al. 1977; Grossman 1977; Lieberman and Wesley 1977; Taylor 1979; Meyer 1981; Summers and Grossman 1982). This experience led in 1978 to creation of a second-generation research robot system with enhanced functional capabilities and a considerably improved programming interface featuring a new programming language, AML. This system became the prototype for the IBM RS 1 Manufacturing System, which is in use in a number of manufacturing and laboratory sites around the world. In addition, a subset of AML is available on the IBM Personal Computer for programming the IBM 7535 Manufacturing System robot.

This paper describes the research version of AML. Although this version of the language is very

The International Journal of Robotics Research
Vol. 1, No. 3, Fall 1982
0278-3649/82/030019-23 $05.00/0

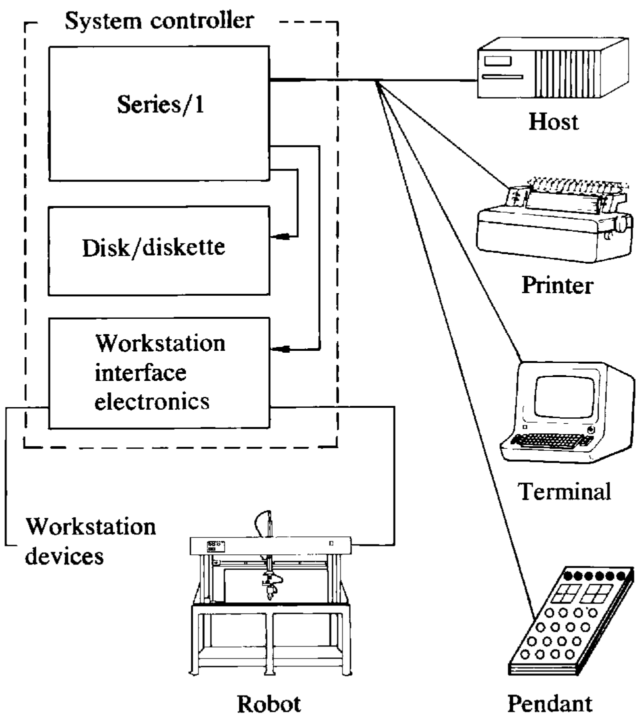

Fig. 1. Research robot system architecture.

similar to the RS 1 version, it contains several experimental enhancements that support robotics research activities. The paper begins with a brief description of the robot system architecture and a statement of the language design objectives. This is followed by a description of the AML language and examples of its use.

2. System Architecture

The architecture of the Research robot system is illustrated in Fig. 1. The central element is a system controller consisting of a Series/1 minicomputer, workstation interface electronics, and a variety of data processing peripherals, including a display terminal, printer, file-storage devices, and a teleprocessing link to the Research computer center. One component of the system controller software provides an interactive programming environment for AML. A second software component performs trajectory planning, motion coordination, sensor monitoring, and other real-time tasks in response to commands from the AML interpreter.

A variety of robot, sensory, and other hardware may be attached to the system controller. The Research robot is a seven-degree-of-freedom hydraulic manipulator whose joints are controlled by analog position servos in the system controller. The robot geometry is basically Cartesian, with three linear motors. Three revolute wrist motors are used to position the gripper in roll, pitch, and yaw coordinates, and a seventh motor is used by the gripper. Sensors typically include force transducers and a light-beam-presence sensor mounted in the fingers, a solid-state television camera, and miscellaneous application-specific sensors, such as "empty" indicators on feeders.

3. Robot Language Design

3.1. Approach

Traditionally, there have been two approaches to the design of robot languages. One approach focuses on the semantics of the robot control system. A syntax is devised to describe how the robot is to move, and then additional language constructs are added, as necessary, to describe sequences of motion, conditional branching, and so on. See, for example, AL (Mujtaba and Goldman 1979), WAVE (Bolles and Paul 1973), PAL (Takase, Paul, and Berg 1981), AUTOPASS (Lieberman and Wesley 1977), and RAPT (Popplestone, Ambler, and Bellos 1978). The advantage of this approach is that it is often possible to provide extremely elegant means of describing various aspects of manipulation.

However, practical robot applications may involve much more than just motion or even motion and sensing. In an inspection application, for example, we found that statements for moving the robot or running the test apparatus constituted less than 10% of the total program. The rest of the code was concerned with such matters as the operator interface, system calibration procedures and calculations, error recovery logic, and data processing for job control. While this may be something of an extreme case, our experience has been that all but the most trivial tasks involve a substantial nonmanipulation component requiring the features of conventional programming languages.

Starting with an existing general-purpose language and extending it by adding the necessary robot semantics is another way of designing robot languages. See, for example, RPL (Park and Burnett 1979), SAIL (Reiser 1976), and MACLISP (Moon 1974). A significant advantage of this approach is that the base programming language is already defined, thus rendering moot many design questions and reducing the amount of user documentation that must be supplied. Also, there may already be a running implementation that can be adapted to robot control. On the other hand, any language, even a "general-purpose" one, embodies design compromises that make it better suited for some purposes than for others.[1]

We decided to follow yet another approach, that of designing a new general-purpose language and extending it with the functions needed to run robots. In surveying the languages that existed in 1977, none had exactly the right mix of features we thought relevant to robotic applications. A new language permitted us the freedom to choose design trade-offs in a meaningful way for the class of applications to be addressed by the language and appropriate for the various users of a robot system.

3.2. Classes of Users

The requirements of users should be a principal concern in the design of any new programming language. In our design discussions, we defined *users* as anyone who must interact with the system software, including machine operators and maintenance personnel as well as application and system programmers. These groups have widely varying requirements and levels of sophistication.

One of our principal objectives was to cover as broad a range as possible with a single programming lanaguage, and to allow the construction of adequate special-purpose interfaces where needed. In our discussions, the following taxonomy proved useful.

Machine operators do not, as a rule, see the programming language itself. Instead, they interact with the system through pushbuttons and lights, simple keyboard responses to terminal prompts, and similar means, as specified by application software. The programming system must permit construction of these interfaces. In addition to support for the necessary input/output (I/O) devices, means for trapping and handling exceptional conditions and console interrupts must also be supplied.

Maintenance personnel typically use canned application programs for most routine maintenance. The interface requirements for these packages are similar to those for machine operators. In addition, however, customer engineers may also generate short test programs and run them interactively. This means that the language must provide a simple, easy-to-use subset for building these programs from system commands and previously defined subroutine libraries. These programs often invoke special "privileged" commands that may compromise system integrity unless access to them is carefully controlled. Consequently, it is also important that the language provide means for enforcing privileges and for "hiding" system primitives from some users, while allowing them to be called from system subroutine libraries.

Application-package users are typically manufacturing engineers with some limited prior exposure to computers. These individuals are interested in specifying assembly, inspection, or materials-movement tasks using simplified programming interfaces (teaching, menu-driven customization, etc.). These interfaces typically either supply data to preexisting application programs or themselves produce simple AML programs consisting largely of subroutine calls.

Specific-application programmers use the language to produce programs for specific applications. These individuals bring a wide range of sophistication to their tasks, which vary from adding

1. As with any simple dichotomy, several exceptional cases do not fit in nicely. For example, APT (IBM Corporation n.d.), a standard but non-general-purpose numerical control language, has been extended for robots by the addition of simple control structures and computational capabilities (Popplestone and Ambler 1978; McDonnell Douglas 1980).

minor embellishments to package-generated programs to producing sophisticated applications with customized operator interfaces, data base links, advanced sensing, and so on. Again, they are likely to make heavy use of prepared subroutine libraries, but they will also tend to write a number of their own subroutines.

Application-package writers write AML programs and subroutine libraries that will be used to produce other programs. Subroutines written for general use often must be more efficient and "robust" than comparable subroutines written for a specific application, since the end users may not always have the ability to modify those elements to meet their needs. Consequently, the package writer will require modern programming language constructs for information hiding, error traps, and so on.

The boundaries among these classes are not at all firm. Indeed, one of the major requirements for the language is that it facilitate ready migration between classes as users become more sophisticated.

3.3. Design Objectives

We designed AML to be a well-structured, semantically powerful interactive language that would be well adapted to robot programming. The central idea was to provide a powerful base language with simple subsets for use by programmers with a wide range of experience. An interpreter implements the base language and defines the primitive operations, such as the rules for manipulating vectors and other "aggregate" objects that are naturally required to describe robot behavior. A major design point of the language was that these rules should be as consistent as possible, with no special-case exceptions. Such a structure provides a ready growth path as programmers and applications grow more sophisticated.

Functional transparency, in the sense that AML subroutines can be written and then used exactly like built-in system commands, was a major design objective. This property allows us to develop functional hierarchies for different user classes and permits us to improve the efficiency of critical functions where required while retaining the advantages of an interpreted implementation.

AML was designed to be an interactive interpretive language because we recognized that debugging and application tuning play a critical role in a programmable robot system. These activities are carried out "on-line"; the robot itself is an important tool in defining critical data points used by the application and as a geometric aid in program debugging. It is often utterly impractical to restart a robot program from the beginning every time a change must be evaluated or a problem diagnosed. Instead, the ability to stop a program, patch it, and continue execution is a critical element in efficiently developing an application.

4. Language Description

4.1. Data Objects

Like other programming languages, AML has constants and variables. A constant or variable has a specific data type, and this data type can be either scalar or aggregate.

4.1.1. Scalar Objects

Scalar object types include *integer, real, string, reference, identifier,* and *label.* The range of values for AML integers and reals is implementation dependent. (Twos-complement arithmetic is assumed.) For the IBM Series/1, an integer has a value in the range (decimal) $-32{,}768$ to $+32{,}767$, corresponding to a 16-bit computer word. Integer constants or expressions that have a value outside this range are coerced to equivalent real values. A real constant has an absolute value in the range 1.0E − 70 to 1.0E70 with approximately six digits of decimal precision.

A *string constant* is a sequence of arbitrary 8-bit characters. On the Series/1, a string constant may be written as a series of EBCDIC characters enclosed by single quotation marks. String values may have characters that range over any of the 256 possible 8-bit values.

References, identifiers, and labels are special forms of data objects. They are each created by special operators and may be used only in restricted contexts. A reference is created by the unary operator &, an identifier by the unary operator $, and labels are created in defining subroutines. Each of these data objects may be passed about as actual parameters to functions.

4.1.2. Aggregate Objects

An *aggregate* is an ordered set of scalar and/or aggregate AML objects. An aggregate need not be homogeneous; that is, it need not contain elements of a single type. The distinction between homogeneous and nonhomogeneous aggregates is only important in realizing certain efficiencies in implementation. The bracketing symbols for constructing an AML aggregate are < and >. Elements of an aggregate are separated from one another by commas. Example of aggregate constants are the following.

```
<1, 2, 3, 4>
<'FEEDER-3', 455, <5.25, 14.75, 2.375>>
<'A', 'B', 'C'>
<>
```

The first example is an aggregate of the first four integers. The second example is an aggregate containing three elements: a string of eight characters, an integer, and another aggregate, itself containing three elements, all of type real. The third example is a homogeneous aggregate of three elements, each of which is a single character string. The final example is an aggregate with no elements. It is called the *null* or *empty* aggregate and is used as the default value for expressions that otherwise have no value.

4.2. Expressions

AML is an expression-oriented rather than a statement-oriented language. This means that every legal construct of the language produces a value that may or may not be used as part of some other expression.

AML expressions are evaluated in left-to-right order. The application of operators in an expression is determined implicitly by the precedence relationships described below or explicitly by the use of parentheses.

Fig. 2. Precedence of AML operators.

1. ? $!
2. aggregate indexing, subroutine parameters
3. &
4. IS = (*left precedence*)
5. unary operators: NOT, +, −
6. | #
7. * / IDIV
8. + −
9. ROTL
10. EQ NE LT LE GT GE VS
11. NOT
12. AND
13. OR XOR
14. OF
15. IS = (*right precedence*)

4.2.1. Primitive Expressions

The simplest expressions in the language are constants and variables. The value of a constant is the constant itself and the value of a variable is the value of the data object that is currently bound to the variable.

Prototypical AML data objects are produced by the primitive expressions INT, REAL, and STRING(*n*), whose values are 0, 0.0, and a string of *n* blanks, respectively.

4.2.2. Scalar Operations

AML supports the usual unary and binary operations on scalar data objects. Except for assignment and data coercion, the binary operators are left associative, and operator precedence similar to that of normal algebra is followed, so that the value of $100-10*3+6$ is 76. Parentheses are used to override the precedence rules, so that the value of $(100-10)*(3+6)$ is 810. A table of precedence rules may be found in Fig. 2.

Arithmetic Operations AML provides the usual operations for integer and real arithmetic:

+	addition
−	subtraction
*	multiplication
/	real division
IDIV	integer division

If both operands are integers and the result can be represented as an integer, then the resulting value is an integer. Otherwise, both operands are coerced to real values and the resulting value is real. The two division operators are exceptions to this rule; their operands are coerced to the appropriate type (real or integer respectively) before the operation is performed. Reals are coerced to integers by rounding to the nearest integer value.

Relational Operators Comparisons between any of the scalar types may be made with the relational operators: EQ, NE, LT, LE, GE, GT, and VS. The first six operators correspond to the usual comparison operators of any programming language. They evaluate to −1 if the relation is "true" and to 0 if the relation is "false." The VS operator evaluates to −1, 0, or 1, depending on whether its first operand is LT, EQ, or GT than its second operand.

Comparisons may be made between strings or between any of the arithmetic data objects. Coercion of an integer to real occurs if the comparison is between operands of different arithmetic types.

Logical Operators AML does not have distinct Boolean data objects, using instead zero values to represent "false" and non-zero values to represent "true." The AML logical operators AND, OR, NOT, and XOR are defined for integer operands and return integer values equal to the logical operations applied to corresponding bits of the operands. The NOT operator performs the ones complement of its operand. The predefined integer constants TRUE and FALSE have values −1 and 0, corresponding to twos-complement integers with all bits on and off respectively.

The extended definition of the Boolean operators to all of the bits in an integer representation provides a concise, though implementation-dependent, means for performing the bit manipulation that is often required in robotics applications.

Another "logical" operator for bit manipulation, ROTL, coerces both operands to integers and then rotates its left operand circularly left by the number of bits equal to the value of its right operand.

Assignment In expression languages such as AML, assignment, =, is treated like any other operator; that is, it evaluates to the value assigned. The precedence of the assignment operator is asymmetric, having a higher precedence to its left than to its right. Thus, an expression

$X = 2 * Y = 3$

assigns 3 to the variable Y then doubles that value and assigns the result, 6, to X. Values are coerced into the type of the target variable before being stored. The value of the assignment expression is the value of the target variable after the assignment is made.

Often, it is useful to specify that a value be coerced to a specific type without carrying out the assignment. The coercion operator, IS, evaluates its left operand to determine a type and then evaluates its right operand and coerces the value to the type of the left operand. For example, the value of

INT IS 10 + 3/4

is 11.

Miscellaneous Scalar Operations The string concatenation operator, |, specifies that its two operands, both strings, are to be concatenated together into a single string.

The reference expression,

& variable

produces a reference to the AML object bound to the variable. Similarly, the expressions

$ *identifier*
$ *operator*

produce primitive objects of type identifier and operator respectively.

The dereference operator, !, undoes the effects of the reference operator. That is, it converts references into values. For example, the AML code sequence

```
X: NEW 3; Y: NEW X; P: NEW &X;
Y = 4 + !P; !P = 10; P = &Y;
```

declares *X* and *Y* to be integer variables with initial value 3 and *P* to be a reference variable initially pointing at *X*'s value (see Section 4.3.3). The value of *Y* is set to 7, the value of *X* is set to 10, and the value of *P* is set to a reference to *Y*'s value.

The type-determination operator, ?, returns a value corresponding to the kind of language object that follows it. Data objects are distinguished by data type and, in the case of variables, by whether or not they are bound to some value. Other language objects include operators and function names.

4.2.3. Aggregate Operations

Aggregate Construction The bracketing symbols, < and >, and the comma separator are the explicit construction operators for aggregates. The evaluation of the expression

< *e1*, *e2*, ... , *ek*>

yields the k-element aggregate containing the values of *el* through *ek*.

Aggregate Concatenation Two aggregates may be concatenated to form a third with the binary operator #. Thus, <1,2,3> # <4,<5,6>> produces the aggregate value <1,2,3,4,<5,6>>.

Replication The replication operator, OF, evaluates its left operand to determine the size of the aggregate to be constructed and evaluates its right operand to determine the value of each of its elements. For example,

```
3 OF 'X'        evaluates to  <'X','X','X'>
2 OF 3 OF .5    evaluates to  <<.5,.5,.5>,<.5,.5,.5>>.
```

Aggregate Assignment Assignments into aggregate variables are performed "element-wise." Thus, if *u* is an *m*-element aggregate variable, the assignment expression,

u = <*vl*, ... ,*vn*>

is equivalent to the aggregate expression

<*u*(1) = *vl*, ... , *u*(*m*) = *vm*   if *n* GE *m*
<*u* (1) = *vl*, ... , *u*(*n*) = *vn*,*u*(*n*+ *l*),..., *u*(*m*)>   if *n* LT *m*.

Similarly,

u = *scalar*

is equivalent to

<*u*(1) = *scalar*, ..., *u*(*m*) = *scalar*>,

and

scalar = <*ul*, ..., *um*>

is equivalent to

scalar = *ul*

if *m* = 1 and is undefined otherwise.

Parallel Assignment Parallel assignment is the extension of the aggregate assignment rules to expressions containing explicitly constructed aggregates for the left-hand side of the assignment operator. The expression

< &*vl*, &*v2*, ... , &*vk*> = *expression*

where & is the reference operator, causes element-wise assignment to each of the variables of the left-hand side according to the rules given above.

Examples of parallel assignment are the following:

```
<&A, &B> = <B, A>;
  -- Exchange values of A and B
<&X, &Y, &Z> = QPOSITION(<1,2,3>);
  -- Break up aggregate of position
  -- values into its component parts.
```

These examples illustrate the two principal uses of parallel assignment: elimination of extraneous temporary variables and decomposition of an aggregate into its constituent elements.

Element Selection AML provides generalized indexing for referencing substuctures in aggregates. The rules are recursively defined as follows.

Expression	*Elements Selected*
$a(i)$	the i'th element of a
$a(i,e2,\ldots,ek)$	$a(i)(e2,\ldots,ek)$
$a(<ul,\ldots,ur>)$	$<a(ul),\ldots,a(ur)>$
$a(<ul,\ldots,ur>,e2,\ldots ek)$	$<a(ul,e2,\ldots,ek),$ $\ldots\ a(ur,e2,\ldots,ek)>$

If a variable data object, XX, has the associated value of

<1,<2,3>,<4,5,6>,<7,<8,9>>>,

the following index operations will correspond to the values shown.

Expression	*Elements Selected*
XX(1)	1
XX(2)	<2,3>
XX(3,2)	5
XX(4)(2)(1)	8
XX(<1,2>)	<1,<2,3>>
XX(<1>)	<1>
XX(<>)	<>

4.2.4. *Mapping of Scalar Operators over Aggregate Operands*

A fundamental and powerful characteristic of AML is the definition of a simple semantics for defining the extension of operators from scalar data objects to aggregate objects. The rules for a binary scalar operator, ∘, are shown below.

Expression	*Value*
$sl \circ s2$	$sl \circ s2$
$sl \circ <vl, \ldots, vk>$	$<sl \circ vl, \ldots, sl \circ vk>$
$<ul, \ldots uk> \circ s2$	$<ul \circ s2, \ldots, uk \circ s2>$
$<ul, \ldots, uk> \circ <vl, \ldots, vk>$	$<ul \circ vl, \ldots, uk \circ vk>$

where

$sl, s2$ = scalars;
$ul, \ldots ,uk$ = arbitrary values; and
$vl, \ldots ,vk$ = arbitrary values.

The rules for unary operators are similar.

Expression	*Value*
$\circ\ sl$	$\circ\ sl$
$\circ <vl, \ldots, vk>$	$<\circ\, v1, \ldots, \circ\, vk>$

The following examples illustrate these rules.

Expression	*Value*
<1,2> + <3,4>	<4,6>
<<1,2>,3> * <4,<5,6>>	<<4,8>, <15,18>>
<6,'POST'> EQ <6,'PEG'>	<−1,0>
'S' GT <'S','T','R','E','E','T'>	<0,0,−1, −1,−1,0>

4.2.5. *Control Expressions*

The control elements of AML include the normal constructs of structured programming, as shown in Fig. 3, and subroutines, which are described in Section 4.3. The rules used to determine the truth of expression values are shown in Fig. 4.

Fig. 3. AML control expressions.

```
If e1 THEN e2 ELSE e3
        has value of e2 if e1 is "true"
        has value of e3 (may be null) if e1 is "false"
WHILE e1 DO e2
        continues to evaluate e2 as long as e1 is "true"
        has value of the last value of e2 (may be null)
REPEAT e1 UNTIL e2
        continues to evaluate e1 until e2 becomes "true"
        has value of the last value of e1
BEGIN e1; ... ; ek END
        has value of ek (may be null if ek missing)
```

Fig. 4. Truth values used in AML conditional expressions: any value not "false" is "true."

Value	Truth Value
Integer	"false" if-and-only-if value is 0
Real	"false" if-and-only-if value is 0.0
Aggregate	"false" if-and-only-if it has no "true" elements

4.3. Subroutines

Subroutines may be used or called from within any expression. Although all subroutines are called the same way, they may exist either as user-written AML code or as primitive elements of the base system.

4.3.1. Subroutine Calls

All subroutines are called in the following way

sname (*expression1*, ... , *expressionk*),

where *sname* is a subroutine name and *expression1* through *expressionk* are AML expressions whose values are to be passed as actual parameters to the subroutine. If no actual parameters are to be passed, the parentheses may be omitted, unless they are needed to distinguish actual parameters from aggregate indices applied to a returned value.

The value of the subroutine call expression is the value passed to a RETURN command executed by the subroutine.

Examples of subroutine calls are the following.

```
MOVE(<1,2>, POSITION + <0.5, 1.7>)
SQRT(X*X+Y*Y+Z*Z)
CABLEAUTOMATION('file',1+SQRT(16.), <15.55, 6.723>)
```

Recursive subroutine calls, that is, calls to subroutines that are already active, are permitted.

4.3.2. Subroutine Declarations

AML subroutines are declared as follows:

```
subrm: SUBR (formal ,..., formalk);
        statement1;
        statement2;
          ...
        statementk;
        END;
```

where *subrnm* is any nonreserved name in the language, *formal1* through *formaln* are formal parameters (described below), and *statement1* through *statementn* may be executable statements (i.e., AML expressions), variable declarations, or subroutine definitions. Optional labels may precede any executable statement.

4.3.3. Variable Declarations

An AML subroutine may contain declaration expressions to define, allocate, and initialize data variables to be used by the subroutine. The form of a declaration is

vname: *mode expression*;

where *vname* is the name of the variable being declared, *mode* specifies how and when storage is to be allocated, and *expression* is any AML expression (except for a declaration) whose value determines the type and initial value of the variable.

New Storage A mode of NEW indicates that a new copy of storage should be allocated for the variable upon each entry into the subroutine and deallocated when the subroutine returns. The *expression* is evaluated on each allocation to determine a type and initial value.

Static Storage Variables declared wtih STATIC mode have their storage allocated and initial values determined during the initial entry into the subroutine. During subsequent entries the variable values are retained from the previous call.

4.3.4. Formal Parameters

Formal parameters may be variable names, reference parameters, or aggregates of formal parameters.

Value Parameters The use of a simple variable name as a formal parameter means that the corresponding actual parameter expression is to be evaluated. When the subroutine is entered, the variable name is bound to temporary storage holding that value.

For example, consider the subroutine

```
SQUAREIT: SUBR(V);
   V=V*V;
   RETURN(V);
   END;
```

If the value of the variable *X* is 3, then SQUAREIT(X) would return 9 but would not modify *X*.

Reference Parameters Typically, reference parameters are used to pass structures whose values are to be modified as a side effect of subroutine calls or that are too large to be passed efficiently as values.

Placing the dereferencing operator, !, in front of a variable name in a formal parameter list specifies that the corresponding parameter is to be passed by reference rather than by value. This means that if the corresponding actual parameter value is a reference, the formal parameter variable name will be bound to the object referred to rather than to the reference value.

For example, consider the subroutine

```
SETV: SUBR(!P,Q);
      P = Q;
      END;
```

Then SETV(&X,3) would set the value of *X* to 3.

The AML interpreter evaluates actual parameters corresponding to reference formals as if they are the left-hand sides of assignment expressions. In other words, simple variable names and index expressions produce references to the corresponding values rather than the values themselves. Consequently,

```
SETV(X,Y)       is equivalent to     X = Y
SETV(&X,Y)      is equivalent to     X = Y
SETV(X(I),3)    is equivalent to     X(I) = 3.
```

Aggregate Parameters A formal parameter may be an aggregate of formal parameters. The aggregate assignment rules are used to associate the elements of the aggregate being passed to the subroutine with the elements of the formal parameter aggregate. For example,

```
GIANT(<'fee','fie','fo','fum'>);
```

where GIANT is defined as

```
GIANT: SUBR(<a,b,c,d>);
    :
    END;
```

would bind a to 'fee', b to 'fie', and so on.

Optional and Excess Parameters Any formal parameters for which there are no corresponding actual parameter values are unbound upon entry to the subroutine. Similarly, any excess actual parameters are ignored.

The ? operator is often used to determine if an optional parameter has been specified in a particular call. For example,

```
OUTPUT_SUBR: SUBR(STUFF,OUT_CHAN);
   OUT_CHAN: NEW IF ?OUT_CHAN
                 THEN OUT_CHAN
                   -- Specified ⇒ use it
```

```
                  ELSE DEFAULT_OUT_CHAN;
                    -- unspecified ⇒ default
        ⋮
        WRITE(OUT_CHAN, STUFF);
        ⋮
        END;
```

PARMS On entry to any subroutine, the variable *PARMS* is always bound to an aggregate consisting of all actual parameter values passed to the subroutine. No implicit dereferencing is done. Thus, in the call SETV(X,3,4), *PARMS* would be bound to the aggregate <&X,3,4>.

4.3.5. Returning Values

Subroutines return values to their calling environment by calling the fundamental subroutine, RETURN:

RETURN(*expression*),

where *expression* is any AML expression. The value produced is returned as the value of the subroutine calling RETURN. If the expression is omitted, the null aggregate is returned. Similarly, if the subroutine evaluation reaches the END statement, the null aggregate is returned.

4.3.6. Subroutine Execution

After all actual parameters have been evaluated, the subroutine is evaluated as follows:

1. The variable *PARMS* is bound to an aggregate of all actual parameters.
2. All formal parameters are bound left-to-right.
3. All local variable declarations, statement labels, and local subroutine definitions are processed in the order they appear in the subroutine body.
4. The executable statements (i.e., the AML expressions) in the subroutine body are evaluated in the order they appear in the subroutine body.

When the subroutine returns, or control passes out of the subroutine because of a nonlocal BRANCH or QUIT command, the following steps are performed:

1. If a CLEANUP command has been executed for this subroutine activation, the subroutine specified by the command is called.
2. All bindings performed in steps 1–3 are undone, and the previous bindings are restored.
3. Any temporary storage allocated for NEW variables and actual parameters is relinquished.
4. Control is passed to the calling AML expression or to a labeled statement, as appropriate.

5. AML Commands

AML commands are simply predefined subroutines that define the semantic functions for robotics, mathematical calculation, I/O, and so on. No syntactic distinction is made between these routines and any other subroutine in the system (see Section 4.3). These functions may exist either as system-defined primitives written in the implementation language of the system or as AML subroutines that are almost always defined as part of the base system.

The semantic environment in which a user's AML program executes is thus defined by the semantics of these routines, together with any additional AML libraries that may have been selected by the user. Once an AML subroutine is loaded into the user workspace, it effectively becomes part of the system. Thus, a GRASP subroutine could be used by a naive user in exactly the same manner as the more primitive MOVE command. This transparency provides a natural growth path for extensions to the system through the use of functional subroutine libraries.

Commands can be classified, roughly, into several categories:

Fundamental subroutines
Calculational subroutines
Robot and sensor I/O commands
System interface commands
Data processing commands

Fig. 5. Fundamental subroutines.

```
                 RETURN(value)
                 BRANCH(label)
                 QUIT(tolevel)
     prev subr = CLEANUP(subr)
     prev subr = ERRTRAP(subr)
           val = APPLY(subr,<p1,...,pk>,level)
    <v1,...,vn = MAP(subr,<<p11,...,p1n>,...,<pk1,...,pkn>>,level)
```

Each category is discussed further below.

The list of AML commands is not complete. We expect continuing research and practical experience to help us identify new and different functions. As this happens, the research AML system should evolve to include additional commands.

New commands usually have their initial implementation as AML subroutines. When greater computational efficiency is required, these routines are often recoded in the underlying system implementation language without changing their functional behavior. The migration of new functions from AML libraries to built-in system commands was anticipated in the original design and has already occurred for several functions in our research system.

5.1. Fundamental Subroutines

Fundamental subroutines (Fig. 5) are an essential part of the AML language definition, although they are called just like any other subroutine. Generally, they are concerned with the flow of control within the language interpreter. For example,

RETURN is used to return a value from a subroutine.
BRANCH is used to cause an unconditional transfer of control.
QUIT is used to cause a return to a specified level of invocation of the interpreter.
CLEANUP is used to set a subroutine exit trap.
ERRTRAP is used to set a subroutine error handler.
APPLY is used to call a subroutine with a program-generated set of arguments.
MAP is used to apply a subroutine element-wise to aggregates.

Several of these subroutines, especially RETURN and BRANCH, are used extensively even by naive users of the language. Others are used primarily by application writers in building subroutine libraries or packages.

5.2. Calculational Subroutines

Figure 6 illustrates subroutines that perform data transformations on their arguments. Several subcategories may be recognized.

First, there are mathematical functions that operate on numbers, such as absolute value, square root, trigonometric subroutines, and so on. They all follow the standard AML mapping rules for aggregate and scalar operations. For example,

the value of
  SQRT(<1.0,4.0,<9.0,16.0>>)
is
  <1.0,2.0,<3.0,4.0>>.
Similarly, the value of
  ATAN(SQRT(<1.,3.>)*10,10)
is <45.,60.>.

A second class of calculational commands is used to extract attribute information about data objects. For example, LENGTH returns the number of characters in a string, and AGGSIZE returns the number of elements in an aggregate.

A third class provides a set of routines for manipulating strings, such as locating and extracting

Fig. 6. Typical calculational subroutines in research AML system. Note that only a subset of available functions is shown.

```
         val = f(num)                 f = ABS, SQRT, SIN,
<v1,...,vn> = f(<n1,...,nn>)              COS, TAN, etc.

           n = AGGSIZE(<a1,...,an>)
           l = LENGTH(string)
<l1,...,ln> = LENGTH(<string1,...,stringn>)

      string = CVTNS(number)
      number = CBTSN(string)
<s1,...,sn> = CVTNS(<num1,...,numn>)
<n1,...,nn> = CVTSN(<string1,...,stringn>)

      string = SUBSTR( string, first, count)
       index = LOCSTR( looked_for, source, start_index)
      result = UNPACK( string,value_template)

< 1,...,n > = IOTA(n)
<a1,...,ak> = SELECT(<m1,...,mn>,<a1,...,an>)

       value = DOT(u,v)
       xpose = TRANSPOSE(m)
       rotmx = EULERROT(<roll,pitch,yaw>)
    <r,p,w> = ROTEULER(<Rx,Ry,Rz>)
```

substrings from a longer string. In addition, it includes routines for packing and unpacking binary data into and out of AML strings and converting between numeric and corresponding string representations.

A fourth class implements various matrix and vector operations. A number of these functions have been borrowed from APL. For example, IOTA (*n*) generates an aggregate of ascending integers from 1 to *n*. Similarly, SELECT extracts a masked subset of elements from an aggregate. Other members of this class implement standard operations (such as inner products and transpose of a matrix) from linear algebra. Still others provide conversions among various ways of representing rotations.

5.3. Robot and Sensor I/O Commands

Figure 7 illustrates subroutines that specialize the system for robotics applications. In accord with the language philosophy, system primitives mainly provide low-level functions; higher-level functions are provided by AML subroutines that use these primitives.

As with calculational commands, several subclasses may be recognized. The first and perhaps most obvious class is concerned with kinematic control of the robot. The primitive MOVE statement specifies a coordinated motion of a specified joint or set of joints to a specified goal or goals. Joints are specified by small integers and goals are specified by numbers giving inches for linear joints and degrees for revolute joints. For example,

```
MOVE(1,10.2);            -- Moves joint 1 to 10.2
MOVE (<1,2>,0);          -- Moves joints 1 and 2 to 0.0
MOVE(<4,5>,<30,90>);     -- Moves joints <4,5> to <30,90>.
```

Each MOVE produces a coordinated trajectory consisting of acceleration, constant-speed travel, deceleration, and an optional wait for final settling. All

Fig. 7. Typical robot and sensor I/O commands.

```
              MOVE(joints,goals,monits,<spd,accel,decel,settle>)
              AMOVE(joints,goals,monits,<spd,accel,decel,settle>)
              DMOVE(joints,displs,monits,<spd,accel,decel,settle>)
      goals = GUIDE(joints)

   oldspeed = SPEED(fraction_of_full_speed)
   oldaccel = ACCEL(fraction_of_full_accel)
   olddecel = DECEL(fraction_of_full_decel)

    oldmove = STOPMOVE
              WAITMOVE

   valueset = QGOAL(joints)
   valueset = QPOSITION(joints)

              FREEZE
              STARTUP
              SHUTDOWN

   valueset = SENSIO(sensorset,template)
  sensornum = DEFIO(iogroup,iotype, format,bitno,nbits,scale,offset)
              DELIO(sensornum)

 monitornum = MONITOR(sensor,testtype,lim1,lim2, freq,subrid)
    flagset = ENDMONITOR(monitorset)
    flagset = REMONITOR(monitorset)
    flagset = QMONITOR(monitorset)
```

joints accelerate and decelerate together and arrive at their final goal simultaneously. The user can control each phase of the trajectory through parameters to the motion statement or through global context commands (SPEED, ACCEL, DECEL). An optional set of sensor monitor numbers may be specified with the motion command (see discussion of MONITOR, below). If any of these monitors "trips" before the motion is complete, the trajectory is halted at that point, and the motion enters its settling phase.

The MOVE command returns once the motion is completed. The AMOVE command returns control as soon as the motion is started, thus allowing the user to overlap computation with motion. WAITMOVE may be used to wait for completion of a motion. Thus, MOVE could (in principle) have been implemented as

```
MOVE: SUBR;
        APPLY($AMOVE,PARMS);
        WAITMOVE;
        END;
```

but was common enough to justify having a separate instance. Similarly, DMOVE, which makes an incremental motion of joints, and GUIDE, which places joints under teleoperator control, could have been implemented in AML but are supplied as system routines for convenience and efficiency.

STOPMOVE causes the currently active motion to be suspended and returns an aggregate of parameters that can be APPLY'd to AMOVE to restart the motion, as illustrated in Section 5.4.

The QGOAL command returns the most recent joint goal(s) computed by the real-time, trajectory-generation software and passed on to the joint controllers, and QPOSITION returns the present actual position of a joint or set of joints.

Fig. 8. Typical system interface commands.

```
          SETBREAK(subrid,stmntno)
oldflag = SINGLESTEP(flag)
val     = BREAK(el,...,ek)
val     = LOAD(filename)
<>      = UNLOAD(object)
oldsubr = KEY(keynum,subrid)
```

Code listing 1. The attention key service.

Code listing 1.

```
ATTN: SUBR;
  OLDMOVE: NEW STOPMOVE;             -- suspend and save
                                     --   old motion, if any
    BREAK('Attention key pressed',EOL,  -- displays all params
          'Arm at ',QPOSITION(ARM),EOL, --   and then enters
          'Current motion is ',      --   read-eval-print loop
            oldmove,EOL);            --   returns when user
                                     --   types "RETURN;"
    APPLY($AMOVE,OLDMOVE);           -- resume motion
    END;
```

A second subclass is concerned with the robot state as a whole. For example, STARTUP instructs the real-time system to turn on arm power, and SHUTDOWN instructs it to turn it off. Similarly, FREEZE instructs the real-time system to prohibit all motion and tightens the acceptable tolerances used by the real-time safety monitoring.

A third subclass deals with sensor inputs and outputs. The DEFIO command defines logical "sensors" and "drivers" as contiguous subfields of 16-bit input and output hardware registers in the system controller and returns small integers that may be used to refer to the entities defined. The user can optionally define scale and offset factors that may be used to convert between hardware values (which are all integers) and floating-point numbers corresponding to engineering units.

The SENSIO command is used to perform I/O of values under direct control of AML programs.

A final subclass allows the user to specify real-time monitoring of sensor values. The MONITOR command tells the real-time system to begin reading a specific sensor or set of sensors at regular time intervals and returns a small integer or set of integers identifying the monitor(s). A specified test, such as checking to see if the value is outside user-specified limits, is performed. If the test succeeds, the monitor is tripped and a flag set. The QMONITOR command is used to query the flag and the REMONITOR command provides an atomic query-with-reset capability. An optional subroutine may be specified with the MONITOR command. If this subroutine is specified, the transition from untripped to tripped state causes the AML program to be interrupted by a call to the specified subroutine.

5.4. System Interface Commands

Figure 8 illustrates subroutines that provide control over the interactive interface of the AML execution environment. Examples include BREAK, which invokes the terminal command processor; LOAD and UNLOAD, which control what objects are in the AML program space; KEY, which allows the user to associate interrupt subroutines with console key events, and so on. For example, the console "attention" key service is provided by an AML subroutine as shown in Code Listing 1.

Fig. 9. Typical data processing I/O subroutines.

Fig. 10. Matrix and vector product.

```
                ALLOCATE(filename,reclen,nrecords)
                CLOSE(iochannel)
                DISPLAY(val1,...,valk)
                EOD(iochannel,recnum)
                ERASE(filename)
   iochannel =  OPEN(filename)
    oldvalue =  POINT(iochannel,value)
                PREFILL(expected_terminal_input)
                PRINT(iochannel,<val1,...,valk>)
    info_agg =  QCHAN(iochannel)
       value =  READ(iochannel,value_template,eoflab)
       value =  WRITE(iochannel,value,eoflab)
                READHOST(host_file_name,local_file_name)
                WRITEHOST(local_file_name,host_file_name)
```

Fig. 10.

```
DOT: SUBR(!U,!V);
    RETURN(IF IS_MATRIX(U) THEN MAP($DOT,<U,&V>)
             ELSE APPLY($+,<U,&V>));
    END;
```

The call

```
KEY(2,$ATTN);
```

associates ATTN with console key number two.

5.5. Data Processing Commands

Figure 9 illustrates subroutines that provide access to the terminal and other data processing peripherals. The basic model for all I/O is a stream-oriented transfer of bytes to or from a logical port called a *channel*. A physical device or file is associated with a channel through the use of an OPEN command; subsequent transfers are performed in an essentially device-independent manner. This permits a great deal of flexibility in constructing application programs and reduces the number of special utilities that must be written. (For example, listing a text file to a terminal or printer can be accomplished by a single COPY utility). It also makes application programs and (more important) system packages relatively independent of the quirks of individual devices. A POINT command is used to position a read/write cursor associated with each logical channel, thus allowing random access to files. An attempt to POINT the cursor of a channel associated with a serial device (such as a printer) causes an error.

6. Examples

A key notion in the design of AML was that specific application programs would make extensive use of previously written subroutine libraries. The examples given in this section are drawn from packages designed for light assembly and materials-handling applications. The subroutines shown have been simplified somewhat for ease of understanding. Enough detail is shown, however, to illustrate the nature of typical AML packages and application codes.

6.1. Matrix and Vector Product

Figure 10 is a routine for implementing the normal matrix and vector product functions of linear alge-

Fig. 11. Cartesian frame of robot's gripper.

```
HANDFRAME: SUBR;
   R: NEW EULERROT(QPOSITION(RPW));-- R = orientation of wrist
     RETURN(<QPOSITION(XYZ)          -- Return <pos of hand,R>
                 - HANDLEN*R(3),R>);
     END;

XYZ:        STATIC <1,2,3>;              -- motor numbers of linear joints
RPW:        STATIC <4,5,6>;              -- motor numbers of wrist joints
HANDLEN: STATIC 8.0;                    -- distance of fingers from wrist
```

bra. Vectors are represented by aggregates of numbers, and matrices are represented by aggregates of vectors corresponding to the rows of the matrix. IS_MATRIX is a simple AML subroutine that uses AGGSIZE to decide whether an aggregate is a vector or a matrix.

6.2. Cartesian Frame of Robot's Gripper

The subroutine shown in Fig. 11 computes a "frame" transformation giving the coordinate system of a defined point in the Research robot's gripper. This frame is represented as an aggregate

<**p**,<**Rx**,**Ry**,**Rz**>>

in which **p** is a vector giving the displacement of a point in the fingers relative to the robot's coordinate system, and <**Rx**,**Ry**,**Rz**> is a matrix giving the orientation of the wrist. (**Rz** gives the direction from the fingers back up to the wrist, and **Ry** gives the direction between the fingers.)

6.3. Centering Grasp

Figure 12 illustrates a typical AML library subroutine for centering the gripper on an object and grasping it with a specified pinching force. The technique used is similar to one developed by R. Paul and used in the Stanford WAVE (Bolles and Paul 1973) and AL (Mujtaba and Goldman 1979) systems.

On entry, the MONITOR command is APPLY'd to an actual parameter list generated by PINCH_FORCE(F) (see Section 6.4). This command instructs the real-time system to begin monitoring the gripper-finger pinch sensors. The variable FMONS is bound to a set of small integers identifying the monitoring activity required. A CLEANUP command is then used to tell the AML interpreter that subroutine CLN is to be called whenever GRASP is exited, ensuring that the monitoring activity will be terminated.

The MOVE command is then used to close the gripper until (1) one of the monitors identified by FMONS is tripped, or (2) the gripper reaches the minimum specified opening.

If one of the force monitors has tripped, then the corresponding finger has contacted the object. The subroutine then moves the robot so that that finger remains in contact while the gripper closes. When both monitors trip, the gripper has been centered on the object.

If the gripper closes to the minimum specified distance without encountering the object or if the final gripper opening exceeds the allowed maximum, appropriate error codes are returned. Otherwise, the string 'OK' is returned to indicate success.

6.4. Sensor-Monitoring Specifications

The declarations illustrated in Fig. 13 are taken from the same AML library that includes GRASP. They are intended to facilitate specification of sensor-monitoring conditions to be used by other subroutines in the package. For example, consider the AML code, shown in Code Listing 2.

APPROACH_MOVE and FINAL_MOVE are AML

Fig. 12. Simplified centering grasp subroutine.

Fig. 13. Typical sensor monitoring specifications, used to supply parameters to high-level motion subroutines.

```
GRASP: SUBR(GRIPPER_OPENING,<MIN_OFS,MAX_OFS>,F);
   TOGO:    NEW REAL;
   FMONS: NEW APPLY($MONITOR,PINCH_FORCE(F));
    CLEANUP($CLN);

    MOVE(GRIPPER,GRIPPER_OPENING+MIN_OFS,FMONS);

    IF QMONITOR(FMONS(1)) EQ O THEN
       BEGIN                                  -- Left finger did not hit.
       IF QMONITOR(FMONS(2)) EQ O THEN RETURN('TOO SMALL');
       TOGO = GRIPPER_OPENING+MIN_OFS-QPOSITION(GRIPPER);
       DMOVE(XYZ#<GRIPPER>,
                (TOGO/2*(HANDFRAME)(2,2))#<TOGO>,FMONS(1));
       END
    ELSE IF QMONITOR(FMONS(2)) EQ O THEN
       BEGIN                                  -- Right finger did not hit.
       TOGO = GRIPPER_OPENING+MIN_OFS-QPOSITION(GRIPPER);
       DMOVE(XYZ#<GRIPPER>,
                (-TOGO/2*HANDFRAME)(2,2))#<TOGO>,FMONS(2));
       END;

    RETURN( IF QPOSITION(GRIPPER)
                    LE GRIPPER_OPENING+MAX_OFFSET THEN'OK'
             ELSE 'TOO BIG');

   CLN: SUBR;                            -- Called whenever GRASP exits
        ENDMONTIOR(FMONS);               --   to terminate force monitors
        END;
    END;
```

Fig. 13.

```
PRESENCE: NEW <LED,1,ON,ON>;        -- Signal event if electric eye
NO_PRESENCE: NEW <LED,1,ON,ON>;     -- (LED) between fingers indicates
                                    -- presence/absence of object

PINCH_FORCE: SUBR(F);          -- F is force value
    RETURN(<<SLP,SRP>,         -- SLP & SRP are predefined force sensors
               1,O,F>>);       --"Trip" monitor if either sensor value
    END;                       -- exceeds range from O to F.

ANY_FORCE: SUBR(F);            -- F is force value
    RETURN(PINCH_FORCE(F)      -- SIDE_FORCE and TIP_FORCE are
                #SIDE_FORCE(F)#TIP_FORCE(F));-- like PINCH_FORCE
    END;

NO_SENSING: NEW <>;
```

Code Listing 2. Code to move the gripper to a specified target position while spacing it to 3.5 in.

```
APPROACH_MOVE(OBJECT_PLACE,<0,0,0>,3.5,PRESENCE);
APPROACH_MOVE(HANDFRAME,<0,0,-1>,3.5,
                         TIP_FORCE(3.0*OZ),SLOWLY);
FINAL_MOVE(HANDFRAME,<0,0,.1>,3.5,NO_SENSING);
GRASP(1.5,<-.1,.1>,16.0*OZ);
```

subroutines that move the gripper to a specified offset relative to a target frame coordinate system; the principal difference between them is that the latter performs more careful nulling of position errors before returning. The sequence above moves the gripper to a specified target position while opening it to 3.5 in. Motion is stopped as soon as the electric eye built into the fingers is obstructed. The gripper is then moved down along the finger axis for another inch until the gripper force sensors detect a force in excess of 3.0 oz. (SLOWLY is an aggregate specifying speed and acceleration parameters.) Finally, the gripper is backed off 0.1 in. and the object between the fingers is grasped.

6.5. Pallet Bookeeping Package

Figure 14 illustrates AML subroutines for use with "palletizing" applications such as the one illustrated in Section 6.6.

The PALLET subroutine is used to create and initialize a data structure for use by other routines in the package. The subroutine examines the optional PLACE parameter to determine the initial coordinate system of the pallet and then returns an aggregate containing the relevant data.

The PALLET_GOAL subroutine returns the coordinate system of the presently indexed pallet element. The INDEX_PALLET subroutine advances the pallet indices. If all pallet positions have been exhausted, it returns 'FULL'; otherwise, it updates the goal value stored in the data structure and returns 'OK'. Note that a different implementation of these subroutines might compute the pallet goal each time it was required, rather than storing it. Such changes would be transparent to the user.

The RESET_PALLET subroutine sets the pallet indices to initial values and (optionally) updates the coordinate system of the pallet.

6.6. Intelligent Pick-and-Place Program

The simplified program shown in Fig. 15 illustrates the use of AML library functions to perform a typical palletizing application. Many details (such as code for operator communication, system calibration, and additional error recovery) that would probably be present in a production program have been omitted. The code shown, however, gives an idea of what an application programmer might have to write.

The problem is to transfer objects from input pallets to output pallets of a different size. Only objects whose diameters are larger than 1.8 in. and smaller than 2.2 in. are to be moved. The input pallets are assumed to be placed manually into the robot's work area. The output pallets are assumed to be fed on a shuttle.

The program starts by declaring variables for the pallets and some I/O points for controlling the shuttle. It then displays a message to the operator asking whether to calibrate the pallet locations, and prompts the operator with the expected reply ('YES'). If calibration is requested, then the appropriate sensing and computation is performed.

The program then begins the main production cycle. First, it calls INDEX_PALLET to increment the indices of the output pallet. If the pallet is full (exhausted), the operator is requested to replace it with an empty pallet. The FREEZE command is used to disable robot motion while the operator is replacing the pallet. This command prevents the system software from changing the robot position goals under any circumstances and tightens the position tolerances used by the real-time safety monitoring.

Fig. 14. Pallet bookkeeping subroutines.

```
PALLET: SUBR(COUNTS,SPACING,PLACE);
   WHERE: NEW NILTRANS;
     IF ?PLACE THEN                  -- Initial value specified
        (IF AGGSIZE(PLACE) EQ 3 THEN &WHERE(1) ELSE &WHERE) = PLACE;
     RETURN(<<1,1>,                  -- Current row, col indices
                 COUNTS,             -- <max rows, max cols>
                 SPACING,            -- <row spacing,col spacing>
                 WHERE,              -- pallet coordinate system
                 WHERE>);            -- current pallet goal

     END;

------------------------------------------------------------------------------------------

PALLET_GOAL: SUBR(!P);
     RETURN(P(5));                   -- Return coord frame of
     END;                            --    currently indexed element

------------------------------------------------------------------------------------------

INDEX_PALLET: SUBR(!P);
     IF P(1,1) LT P(2,1) THEN        -- Is current row full?
        P(1,1) = P(1,1) + 1          -- No, increment row index
     ELSE IF P(1,2) LT P(2,2) THEN   -- Row full, another col to go?
        P(1) = <1,P(1,2)+1>          -- Yes, reset row, incr column.
     ELSE RETURN('EXHAUSTED');       -- No more columns, return "full".

     -- Here, p(1) = <current row index, current col index>.
     -- Compute an updated coordinate frame for PALLET_GOAL value.

     P(5,1) = DOT((P(1)-1)*P(3)#<0>,P(4,2))+P(4,1);
     RETURN('OK');                   -- Return success.
     END;

------------------------------------------------------------------------------------------

RESET_PALLET: SUBR(!P,NEW_INDICES,NEW_LOC);
     IF ?NEW_LOC THEN             -- Is new pallet loction specified?
        IF AGGSIZE(NEW_XF) EQ 3 THEN   -- Yes, only vector spec'd?
           P(4,1) = NEW_LOC
        ELSE P(4) = NEW_LOC;
     IF ?NEW_INDICES THEN         -- If new index values specified then
                                  -- bounds check them and save in P(1)
        IF NEW_INDICES GT P(2) OR NEW_INDICES LE O THEN
           RETURN('ILLEGAL_BOUNDS');
        ELSE P(1) = NEW_INDICES       -- If new index values not specified, then
                                      -- assume <1,1> as default.

     -- Here, p(1) = <current row index, current col index>.
     -- Compute an updated coordinate frame for PALLET_GOAL value.

     P(5,1) = DOT((P(1)-1)*P(3)#<O>,P(4,2))+P(4,1)

     RETURN('OK');                  -- Return success.
     END;
```

Fig. 15. Intelligent pick-and-place application.

```
PICKPUT: SUBR;

  PALLET_1: NEW PALLET(<4,5>,<2.5,2.5>,<0,-5,4>);
  PALLET_2: NEW PALLET(<3,4>,<3.0,2.5>,<0,15,4>);

  P1_SHUTTLE:        NEW DEFIO(25,-1,2,0,1);  -- Shuttle control output
  P1_PRESENT:        NEW DEFIO(21,0,2,0,1);   -- Pallet present input
  P1_SHUTTLE_HOME:   NEW DEFIO(21,0,2,1,1);   -- Shuttle at "home" input

  IF ASK_YESNO('Calibrate the pallets?','YES') THEN
     BEGIN
     CALIBRATE_PALLET_COORDINATES(PALLET_1); -- not shown
     CALIBRATE_PALLET_COORDINATES(PALLET_2); -- not shown
     END;

  MAIN_LOOP:
    --
    -- First, increment the destination pallet indices.
    --
    IF INDEX_PALLET(PALLET_2) EQ 'EXHAUSTED' THEN
       BEGIN                              -- pallet 2 needs replacing
       FREEZE;                            -- Freeze robot motion
       DISPLAY('REPLACE PALLET NUMBER 2',EOL, -- Inform operator
               'PRESS THE "RUN" BUTTON WHEN READY',EOL);
       WAITRUN;                           -- Wait for "run" mode
       RESET_PALLET(PALLET_2);            -- Reset pallet indices
       END;

    -- Next, increment the destination pallet indices. Note that
    -- the error recovery logic branches here if a grasp fails.

  PICK_SEQUENCE_START:
    IF INDEX_PALLET(PALLET_1) EQ 'EXHAUSTED' THEN
       BEGIN

       -- Here, all index positions in pallet 1 have been used.

       SENSIO(P1_SHUTTLE,1);             -- Start shuttle out
       WAITI(P1_PRESENT,0);              -- Wait for pallet to be removed
       WAITI(P1_PRESENT,1);              -- Wait for new pallet to be present
       SENSIO(P1_SHUTTLE,0);             -- Start shuttle back in
       WAITI(P1_SHUTTLE_HOME,1);         -- Wait for shuttle locked home sensor
       RESET_PALLET(PALLET_1);           -- Reset the pallet indices.
       END;

    APPROACH_MOVE(PALLET_GOAL(PALLET_1),<0,0,3>,3.0);
    FINAL_MOVE(PALLET_GOAL(PALLET_1),<0,0,1.2>);

    IF GRASP(2.0,<-.2,.2>,5.0*OZS) NE 'OK' THEN
       BEGIN                     -- Grasp failed. Do error recovery.
       MOVE(GRIPPER,3.0);        -- Open gripper
       APPROACH_MOVE(PALLET_GOAL(PALLET_1),<0,0,3>); -- Move clear
       BRANCH(PICKUP_SEQUENCE_START); -- Go try next position
       END;

    APPROACH_MOVE(PALLET_GOAL(PALLET_1),<0,0,3>);
    APPROACH_MOVE(PALLET_GOAL(PALLET_2),<0,0,3>);
    FINAL_MOVE(PALLET_GOAL(PALLET_2),<0,0,0>,<>,TIP_FORCE(3.0*OZS));
    MOVE(GRIPPER,3.0);
    APPROACH_MOVE(PALLET_GOAL(PALLET_2),<0,0,3>);
    BRANCH(MAIN_LOOP);
    END;
```

The AML program busy-waits for the operator to push the "run" button to remove the robot from FREEZE mode and then calls RESET_PALLET to set the pallet indices back to <1,1>.

The input pallet indices are then incremented. If all pallet positions have been exhausted, then a control sequence is generated to activate the shuttle. Once a new pallet has been fetched, the pallet indices are reset.

The program then moves the gripper to the current input pallet goal and grasps the object. If the grasping sequence fails, the program opens the gripper, moves the robot clear, and branches back to try the next pallet position. Real production programs would possibly use more sensing on the approach moves and would probably include additional program logic to detect repeated grasping failures and initiate some other error recovery.

Once the object has been successfully grasped, the program moves it to the destination pallet, releases it, and branches back to restart the cycle. Note that it would be relatively simple to add additional processing or inspection steps, such as using the gripper force sensor to weigh the objects being moved.

7. Experience and Conclusions

This paper has described an enhanced version of the AML language presently in use at the IBM T. J. Watson Research Center. Although the language described is very similar to the language used to control the IBM RS 1 Manufacturing System, it has been enhanced to support our present research activities. Since these enhancements are experimental, they may never be part of any IBM product.

Earlier versions of AML have been operational for four years at the IBM Research Center and at selected IBM manufacturing sites around the world. More recently, the RS 1 version of the language has been used in a limited number of non-IBM sites as well. The language has been used to program a number of applications, including assembly, intelligent materials movement, inspection, and testing. It has been used successfully by a variety of people, including researchers, professional programmers, manufacturing engineers, and (through very simplified interfaces) direct manufacturing personnel.

The experience gained has shown that AML has the power and flexibility to allow development of simple programs in a matter of hours. More sophisticated applications, complete with operator interfaces, calibration packages, and data base links, have been developed in periods ranging from a few days to several months. In all cases, the interactive debugging facilities of the language system have proved especially important.

We have observed that the availability of canned subroutines for difficult or sophisticated program elements significantly reduces the time required to produce a working program. Users often load packages of such subroutines and call them exactly like system primitives, thus confirming a major design assumption of the language. However, many users are also willing to customize previously defined packages to meet their particular needs. On several occasions, manufacturing engineers with limited prior programming experience have gradually developed considerable programming skill, in part by copying the coding style exhibited by expertly written AML programs. This experience confirms our original design hypothesis that a robot language should be a powerful and modular general-purpose programming language that can be used by a continuum of users.

Acknowledgments

We would like to thank our colleagues in the Automation Research Project, in IBM Advanced Manufacturing Systems, and at many IBM manufacturing sites for their helpful suggestions and good-spirited use of evolving versions of the language. We are grateful to Peter Will, who was instigator, protector, and champion of the initial effort. Dave Grossman deserves special thanks for defining the functional requirements of the initial command set and the real-time control system, and for his service as an invaluable adviser and commentator on the AML design. Finally, we gratefully acknowledge the excellent work of Steve Buckley of IBM Advanced Manufacturing Systems, who produced the RS 1 implementation of the AML language and made several valuable enhancements while doing so, and of Don Tolchin, who produced the communications packages and the initial implementation of the real-time robot control system.

References

Blasgen, M., and Darringer, J. 1977. MAPLE: A high level language for research in mechanical assembly. Rept. RC-5606. Yorktown Heights, N.Y.: IBM T. J. Watson Research Center.

Bolles, R., and Paul, R. 1973. The use of sensory feedback in a programmable assembly system. AIM-220. Stanford, Calif: Stanford University Artificial Intelligence Laboratory.

Evans, R., et al. 1977. Software system for a computer controlled manipulator. Rept. RC-6210. Yorktown Heights, N.Y.: IBM T. J. Watson Research Center.

Grossman, D. 1977. Programming a computer controlled manipulator by guiding through the motions. Rept. RC-6393. Yorktown Heights, N.Y.: IBM T. J. Watson Research Center.

Grossman, D., and Taylor, R. 1978. Interactive generation of object models with a manipulator. *IEEE Trans. Syst. Man Cybern.* SMC-8: 667.

IBM Corporation. n.d. *System/370 APT-AC numerical control program reference manual.* SH20-1414. White Plains, N.Y.: IBM Corporation.

Lieberman, L., and Wesley, M. 1977. AUTOPASS: An automatic programming system for computer controlled mechanical assembly. *IBM J. Res. Development* 21: 321.

Lozano-Perez, T., and Winston, P. 1977. LAMA: A language for automatic mechanical assembly. *Proc. 5th Int. Joint Conf. Artificial Intell.* Cambridge, Mass.: M.I.T. Artificial Intelligence Laboratory.

McDonnell Douglas. 1980. Robotic system for aerospace batch manufacturing. St. Louis: McDonnell Douglas Corporation.

Meyer, J. 1981. An emulation system for programmable sensory robots. *IBM J. Res. Development* 25: 955.

Moon, D. 1974. *MACLISP reference manual, version O.* Cambridge, Mass.: M.I.T. Laboratory for Computer Science.

Mujtaba, S., and Goldman, R. 1979. AL user's manual. AIM-323. Stanford, Calif.: Stanford University Artificial Intelligence Laboratory.

Olivetti. n.d. *Robot technology at Olivetti / The SIGMA system.* Torino, Italy: Olivetti Sistemi per l'Automazione.

Park, W., and Burnett, D. 1979. An interactive incremental compiler for more productive programming of computer-controlled industrial robots and flexible automation systems. *Proc. 9th Int. Symp. Industrial Robots.* Dearborn, Mich.: Society of Manufacturing Engineers.

Popplestone, R. J., Ambler, A. P., and Bellos, I. 1978. RAPT: A language for describing assemblies. *Industrial Robot* 5: 131.

Reiser, J. F., ed. 1976. SAIL. AIM-289. Stanford, Calif.: Stanford University Artificial Intelligence Laboratory.

Summers, P. D., and Grossman, D. D. 1982. XPROBE: An experimental system for programming robots by example. Rept. RC-9082. Yorktown Heights, N.Y.: IBM T. J. Watson Research Center.

Takase, K., Paul, R., and Berg, E. 1981. A structured approach to robot programming and teaching. *Proc. COMPSAC79.* New York: IEEE Computer Society.

Taylor, R. H. 1976. A synthesis of manipulator control programs from task-level specifications. AIM-282. Stanford, Calif.: Stanford University Artificial Intelligence Laboratory.

Taylor, R. 1979. Planning and execution of straight line manipulator trajectories. *IBM J. Res. Development* 23: 424.

Unimation. 1979 (Feb.). User's guide to VAL, a robot programming and control system. Version 11. Danbury, Conn.: Unimation Inc.

VanderBrug, G. J. 1981. RAIL: A language for vision and robotics. *Proc. COMPSAC81.* New York: IEEE Computer Society.

Will, P., and Grossman, D. 1975. An experimental system for computer controlled mechanical assembly. *IEEE Trans. Comput.* C-24: 879.