AML/X: A Programming Language for Design and Manufacturing

Lee R. Nackman
Mark A. Lavin
Russell H. Taylor
Walter C. Dietrich, Jr.
David D. Grossman

Manufacturing Research Department
IBM Thomas J. Watson Research Center
Yorktown Heights, New York, 10598

Abstract

AML/X is a modern general purpose high level programming language, aimed at applications in manufacturing and computer aided design. It includes features for both conventional and object-oriented programming. The AML/X interpreter is implemented in C and has been ported to IBM 370, Motorola 68000, and IBM PC hardware, running under CMS, UNIX, XENIX, and DOS. This paper describes the rationale for AML/X and gives an overview of the language itself.

1.0 Introduction

Architects of industrial automation systems make a trade-off between two goals: ease of use and flexibility. A decade ago, when computers were far more expensive, these goals were seen as competing alternatives.

Historically, many industrial automation systems limited their flexibility to the minimum requirements of some (hopefully) large class of users, but no more. Those users then specified applications with a minimal language, which was simple and easy to learn. An example of this approach in robot programming was teaching by showing, in which the user manually moves a robot through a sequence of motions, recording them for later replay[1]. The disadvantage is the lack of a growth path: once the user's needs exceed the flexibility provided, the system becomes ineffective.

Clearly, there is significant benefit in providing systems with much greater flexibility. Generally, this flexibility is achieved by deferring certain choices until as late as possible. For example, a robot achieves more flexibility than fixed automation by deferring the choice of motions from the time the equipment is designed until it is actually used.

Flexibility gained by deferring choices implies the need for a much richer *language* in which to specify the choices to be made. The challenge for the system architect is to layer this language so that ease of use is not sacrificed. As computers decline in cost, this approach of layering is becoming an increasingly practical means of combining flexibility with ease of use.

One begins at the bottom layer with as much flexibility as one can afford (more on this later) and a language for specifying the remaining choices. The language must be expressive enough to allow the most sophisticated users to take advantage of the flexibility included. Perhaps less obvious, but equally important, the language must provide mechanisms for composing higher layers, each essentially a new language, with less flexibility and fewer remaining choices than the preceding layer. The highest layers, those with the least flexibility, provide ease of use for unsophisticated users along with potential growth to lower, more flexible layers.

This is the context in which we have designed AML/X, a general purpose programming language tailored for use in manufacturing and computer aided design. Its roots are in AML[2], a programming language originally developed for use in a research robot system[3] and subsequently made available as the programming language of the IBM 7565 Manufacturing System. AML also saw use as the base language for AML/V, an industrial machine vision programming system built as an extension to the research robot system[4]. AML/X is the result of a major redesign of AML and is part of the programming environment for our research activities in robotics[5], machine vision, and computer aided design. As such, it is likely to continue to evolve. It is also the basis for AML/2, the programming language for IBM's new 7575 and 7576 Manufacturing Systems.

This paper describes the rationale for the design and implementation of AML/X, provides a brief overview of the language, and illustrates its use. For brevity and clarity this paper emphasizes the use of AML/X in robotics; however, AML/X is actively used in our research in machine vision, workcell layout, and computer aided design. The next section outlines our objectives and their influence on the design of AML/X. This is followed by an overview of the basic structure and facilities of AML/X, just enough to provide the flavor of the language and to be able to follow subsequent examples. (A detailed description of AML/X is available in [6].)

2.0 Design Rationale

2.1 Guiding Principles

Designing a programming language requires balancing the many conflicting requirements of the anticipated user community. To help achieve this balance we have tried to follow a few broad, general principles, against which specific design decisions could be evaluated.

First among these is Hoare's dictum[7] that the language designer's task is one of "consolidation, not innovation." In keeping with that principle, we have not introduced any radically new constructs in AML/X. Instead, we have chosen from among constructs and ideas that have been tested in other languages and have tried to integrate them into a consistent, coherent whole. LISP, APL, and SMALLTALK have had an especially strong influence.

The second guiding principle is to prefer general purpose constructs to those meeting specific, limited, application needs. A corollary is that it must then be easy to provide layers that adapt general purpose features to meet the specific needs. For example, AML/X's interactive debugging tools are a small extension of a very general exception handling mechanism. These principles keep the language a coherent whole, rather than an accumulation of features. In practice, this has been very difficult. When an important user requests a specific new feature, it is hard to say no.

The third principle is orthogonality, which demands that separate features be separate, that is, that the legality and meaning of the use of a construct should be independent of its use in combination with some other construct. Orthogonality aids in achieving generality, but also makes it easier to write obscure programs. It can also be difficult to achieve in a practical implementation. For these reasons, we have occasionally violated orthogonality, but not without careful thought.

These three principles are helpful, but they do not substitute for an understanding of the anticipated applications of the language.

2.2 Language Design Criteria

We have designed AML/X keeping in mind three major areas of use: robot programming, machine vision programming, and computer-aided design (primarily of mechanical objects and assemblies, see e.g. [8]). An analysis of the requirements of robot programming and a survey of existing robot programming systems appears in [9]. Requirements for machine vision programming and an abstract language for vision programming (which could be embodied in many ways) are described in [10]. Some issues in programming languages for CAD systems and a particular language design are discussed in [11].

In all three areas, flexibility has historically been sacrificed, primarily to achieve ease of use. As a consequence, existing systems rarely exploit the intrinsic similarities between these domains. For example, all three domains deal with mechanical objects, assemblies, geometric algorithms, and transformations in 2 and 3 dimensions.

In the future, applications will expand in all three domains, and there will be increased need in industry to integrate them. It is therefore desirable to have a single language that can effectively deal with robotics, vision, and CAD.

In the following subsections we discuss the implications of the intended uses of AML/X.

Classes of Users: Roughly speaking, we anticipate three classes of users. *End users*, such as manufacturing engineers or mechanical designers, are the people who specify the operations to be carried out by the automation or CAD system. They typically have little programming skill, but need to be able to "chain together" sequences of pre-existing, relatively high-level commands, possibly including some simple control flow. *Application developers* write application packages, programs that provide facilities for use by a small class of end users. In a sense, an application package decreases generality while increasing ease of use for a particular class of users. Application developers typically have some programming skill and a deep knowledge of the application area. Often they produce new application packages by building a layer on top of an existing, more general program. *Application development environment developers* write the (typically) large systems application developers use. Robot programming and CAD systems are examples. They need both very high programming skill and a reasonably deep knowledge of the application area. This mix of users and the way they work has had an important influence on the language design.

Most significantly, the language must be a modern, general purpose language well-suited to developing large, layered systems. It must support the definition and use of abstract data types and have facilities for manipulating strings, complex data structures, and symbolic information. A general exception handling mechanism is essential so that an application can "catch" exceptions from lower layers, thus avoiding the incomprehensible error messages to end users that would otherwise result.

At the same time, the language must be simple, or at least have simple subsets, so that it is accessible to end users. However, as Hoare has observed[7], subsetting is simpler said than done, because an errant program can invoke some language feature outside of the subset. It is therefore important to be able to selectively disable or hide portions of the language. Moreover, the use of subsets increases the need for consistency and lack of special cases to provide an easy growth path for users who become more sophisticated over time.

Robot Programming: Our experience with several generations of robot system[3, 12, 13] has confirmed that specification of manipulator motion represents only a small, though very important, proportion of the total code required for a working robot application. Other components include I/O for auxiliary devices and communications, operator interfaces, calibration and setup routines, bookkeeping, access to manufacturing data bases, etc. The failure of special-purpose manipulation languages such as AL [14] to provide adequate support for these other components has led to renewed interest [e.g., 15] in the use of standard general-purpose languages for robot programming. The approach in AML, carried over to AML/X, was to design a new general-purpose language whose design trade-offs make it convenient for automation programming.

Robotic applications *do* have many special characteristics. Manipulator motion specifications and calibration packages rely heavily on geometric calculations, and means must be provided for expressing these conveniently. Most applications require some level of concurrency. Application programs are usually debugged (and often written) "on-line" and are hard to restart, thus making support for interactive programming especially important.

The structure of programs is often rather different from that found in other, more "algorithmic" domains. Generally, the main line of an application program is quite straightforward, and consists of little more than a sequence of commands, with most of the useful work being done as a "side effect". Unfortunately, the vagaries of the physical world [16] cannot be ignored. Our experience has been that robust programs can easily have three to five times as much error testing and recovery as main line code. Often, the error recovery actions are both context dependent and safety related. For example, it may be appropriate to freeze manipulator motion if a gripper feedback sensor fails *unless* the hand happens to be in a furnace at the time. These considerations have led us to place special emphasis on powerful exception handling mechanisms.

Machine Vision: Many of the considerations for robot programming also apply to machine vision [4, 10]. Indeed, one of our objectives was to promote better integration of robot and vision programs. The large amount of data that must be handled in many vision programs means that it is especially important to provide data representations and execution primitives with efficient low-level implementations while still providing great expressive power at higher levels. Beyond this, the language should permit suitable interfaces to special-purpose computational hardware.

Computer-aided Design: The requirements for computer-aided design overlap those for robotics and vision. We require good support for layering, support for interaction at high levels, efficient low-level code execution, and extensibility. Requirements that are especially acute in CAD applications include support for self-describing objects, efficient and accurate numerical computation, and the ability to build up extremely complex data structures.

Other criteria: Since we were generally pleased with our experience with AML, we decided early in our design effort to maintain its general flavor, but not to require strict upward compatibility. The result is that most nontrivial AML programs will not run unaltered on AML/X. One reason for this decision was to enable us to make several changes for future compilabilty.

Although we wanted to take advantage of object-oriented language constructs to provide language extensibility, "layering", and support for modular programming, we also wanted to preserve the procedural style that many of our users were accustomed to. We also felt that it was essential to provide good interfaces to other languages to provide "maturity" (e.g., mathematical subroutine packages), to allow for efficient low-level execution (our first implementation is an interpreter), and to allow us to integrate large and diverse subsystems.

3.0 Language Description

This section describes the basic entities and operations available to the AML/X programmer. We begin with some definitions:

objects These are the entities that can be directly manipulated by AML/X.

values The value of an object is the interpretation of its contents; it is meaningful to distinguish between an object and its value, since the value of an object can be changed.

types Each AML/X object has a type, which defines the set of values that that object can have. AML/X includes several numeric types (INT, REAL, etc.), types for character- and bit- strings, and more complicated types.

variables These are symbolic names (e.g., Foo, VAR_NAME, x) by which a programmer can refer to objects in AML/X; the object to which a variable refers is called its *binding*. We also speak of the "type" and "value" of a variable, by which we mean the type and value of its binding.

As in most other programming languages, variables are a key feature of AML/X. They can be manipulated in several ways:

evaluation Retrieving the binding of a variable, that is, the object to which it refers symbolically.

assignment Changing the value of the object bound to a variable. Since the type of an object is fixed, assignment cannot change the type of a variable.

binding Changing the binding of a variable, that is, causing it to be bound to a new object. Unlike assignment, there is no restriction about the type of the new object to which the variable is bound.

Expressions are combinations of variables, constants, and operators that are *evaluated* to produce new objects and, in the case of assignment, to change the values of existing objects. AML/X is a so-called *expression-oriented* language, in that all program execution can be described as expression evaluation.

3.1 Data Objects and Operators

AML/X has the usual complement of basic data objects and operators necessary for "language-hood". These, and a few other objects and operators are described cursorily in this section. All AML/X code is written in THIS TYPE FONT so that it stands out from the rest of the text without resorting to the use of various awkward quote marks. In code examples, we have used upper case for usage that is required (e.g., reserved words) and lower case elsewhere (e.g., identifier names).

Numeric Objects and Operators: AML/X has four kinds of numeric objects: INT and LONG_INT, which correspond to 16- and 32-bit integers, and REAL and LONG_REAL, which correspond to single- and double-precision floating point numbers respectively. The usual binary arithmetic operations addition (+), subtraction (-), multiplication (*), division (DIV), and exponentiation (**) are provided.

String Objects and Operators: AML/X provides both BIT and CHAR strings, consisting of zero or more bits or bytes, which can be manipulated collectively or individually. The usual operations for manipulating both CHAR and BIT strings, including current and maximum length, lexicographic comparison, concatenation, and selection, are provided. The standard bitwise logical operations AND, OR, XOR, logical and arithmetic shifts, rotate, and (unary) NOT are provided for BIT strings.

Boolean Objects and Operators: BOOLEAN objects can have two values, which are denoted by the reserved words TRUE and FALSE. The standard Boolean operations AND, OR, XOR, and (unary) NOT are available in AML/X. There are also two "short-circuit" Boolean operators, CAND and COR, which do not evaluate their right operands unless it is necessary. Thus, if x is

#	Concatenate two aggregates to form a new aggregate
OF	Replicate an aggregate a specified number of times
IOTA	Make an aggregate containing a sequence of integers
ISAGG	Is an object an aggregate?
AGGSIZE	Number of elements in an aggregate?
MAP	Distribute an operator or subroutine over arguments
REDUCE	"Place" an operator between successive elements of an aggregate and evaluate
SCAN	Like REDUCE but return an aggregate of partial results
ANY	Are any elements of an aggregate TRUE?
ALL	Are all elements of an aggregate TRUE?
EQUAL	Are the arguments equal in both structure and value?
COMPRESS	Select elements of an aggregate using an aggregate of BOOLEANs as a mask
AGGLOC	Locate a target aggregate within another aggregate

Figure 1. Aggregate built-in operators and subroutines.

zero, x NE 0 CAND y / x GT 10 evaluates to FALSE and does not divide by zero.

Symbol Objects: SYMBOL objects are variables that are not evaluated (like quoted atoms in LISP) and can therefore be used as names. The notation $name defines a SYMBOL object.

Reference Objects: An object of type REF "points at" an AML/X object, which is called the *referand* of the REF. REF objects can be created by either using the & operator or by calling REF. The referand of a REF object is obtained by using the dereferencing operator (!).

DEFAULT: The AML/X object DEFAULT is used in situations where an object is needed but no particular one is required. For example, the return value of a function called only for its side effects is DEFAULT.

Type Objects and Operators: All AML/X objects have a type, which can be determined using the typeof operator (?). For example, the result of the expression ?3 is the type INT.

3.2 Aggregate Data Objects

Aggregates are one of the most important features of AML/X since they are its most basic data grouping mechanism. An *aggregate* is exactly what its name implies: a collection of objects that can be treated as one. An AML/X object that is not an aggregate, such as an INT or CHAR, is called a *scalar* object.

Creating Aggregates: Aggregates are created using the *aggregation operator* pair, < >, to form a single AML/X object from an explicit list of other AML/X objects. The general form

```
< e1, ..., ek, ..., en >
```

creates an n-element aggregate containing the elements ek. The important points to note are that: (1) any AML/X object can be an aggregate element; (2) elements of an aggregate need not all be the same type; and (3) each element is the object that is the result of evaluating an (arbitrary) expression that specifies the element. Once an aggregate is created, its size cannot be changed. Figure 1 lists some of the operators and built-in subroutines for working with aggregates.

Aggregate Subscripts: Individual aggregate elements can be referred to by numeric indices or *subscripts*. If a is an aggregate, the expression a(s) is a *subscripted aggregate reference*; we also say that a is being *applied* to s. Elements of multi-dimensional aggregates (where some elements are themselves aggregates) can be referenced by multiple subscripts. For example, a(3)(2) refers to the second element of the third element of a. Since this syntax is awkward, multiple subscripts can be elided, so that a(3)(2) can be written as a(3, 2).

The most general case of subscripting occurs with multi-dimensional aggregates subscripted by arbitrary combinations of scalar and aggregate subscripts. The rules are simple. Suppose

a	is an arbitrary aggregate
i	is a positive integer
s	is an aggregate of positive integers or aggregates
..rest..	denotes the "rest" of a list of subscripts (possibly empty).

Then, the meaning of any subscript can be determined by applying the following rules recursively until all subscripts are reduced to scalar values:

Expression	*Equivalent*
a(i,..rest..)	(a(i)) (..rest..)
a(s,..rest..)	<...,a(s(k),..rest..),...>
a(DEFAULT,..rest..)	<...,a(k,..rest..),...>

A few examples will help to give an intuitive understanding of what these rules mean. We'll start with the 3x4 matrix m defined by

```
m: NEW < < 1,  2,  3,  4 >,
         < 5,  6,  7,  8 >,
         < 9, 10, 11, 12 > >;
```

The NEW keyword indicates that this is a variable declaration.

It is simple to use aggregate subscripts to refer to rows and columns of m. If i is a scalar, then m(i) refers to the i-th row of m and m(DEFAULT, i) refers to the i-th column of m. The latter can be written more succinctly as m(, i). In general, any "missing" subscript is treated as if DEFAULT had been specified. The "missing" subscript notation is convenient for denoting entire rows or columns. Selected rows or columns can be referred to by using aggregate subscripts, as illustrated below:

```
m(<1, 3>)          ## 1st and third rows
m(, IOTA(2,4))     ## 2nd through 4th cols
m(, <4, 1>)        ## 4th and 1st columns
m(<1,2>, <1,2>)    ## Upper left 2x2
```

- `IF cond THEN e1 [ELSE e2]`
  evaluates e1 if cond is TRUE and to e2 otherwise; if e2 is ommitted, DEFAULT is used
- `WHILE cond DO expr`
  continues to evaluate expr while cond is TRUE; the result is the result of the last evaluation of expr, or DEFAULT if none
- `WHILE cond DO COLLECT expr`
  continues to evaluate expr while cond is TRUE; the result is an aggregate of the successive results of evaluating expr, or the empty aggregate if none
- `REPEAT expr UNTIL cont`
  continues to evaluate expr until cond becomes TRUE; the result is the result of the last evaluation of expr, or DEFAULT if none
- `REPEAT COLLECT expr UNTIL cont`
  continues to evaluate expr until cond becomes TRUE; the result is an aggregate of the successive results of evaluating expr, or the empty aggregate if none
- `BEGIN e1; ... ek; END`
  the expressions ei are successively evaluated; the result is the result of evaluating ek.
- `SELECT expr CASE c1 THEN e1 ... CASE ck THEN ek [OTHERWISE oexpr] END`
  The result of evaluating expr is compared against successive ei; if there is a match, the result is ci; if no match the result is oexpr if present, and DEFAULT otherwise

Figure 2. Control Operators

Operator Mapping: Operators extend to aggregate operands through a set of *mapping rules* (adopted with slight alteration from AML[2]) which are an abstract form of the distributive law of arithmetic. Suppose

s	is any scalar object
<u1,...,un>	is an n-element aggregate
<v1,...,vn>	is an n-element aggregate
op	is an AML/X operator

Then, the operator mapping rules are:

Expression	***Equivalent***
op <u1,...,un>	<op u1,...,op un>
s op <v1,...,vn>	<s op v1,...,s op vn>
<u1,...,un> op s	<u1 op s,...,un op s>
<u1,...,un> op <v1,...,vn>	<u1 op v1,...,un op vn>

These mapping rules apply recursively to multi-dimensional aggregates.

A key point about the mapping rules is that they work in a uniform way for most AML/X operators. In particular, "parallel assignment" can be written in the form <v1,...,vk> = expression.

3.3 Control Operators

AML/X supports control flow constructs typical of a structured programming language. These are summarized in Figure 2. Since AML/X is an expression-oriented language, all control flow constructs are operators that yield a result like any other operator, as illustrated by the expression

```
move(WHILE ask('More?') DO COLLECT teach())
```

which creates an aggregate of (presumably) points returned by successive calls to teach and passes it to the subroutine move.

3.4 Expression Evaluation

AML/X is an *expression-oriented* language, which means that *every* AML/X program construct is an expression. Thus, expression evaluation is the fundamental computational process in AML/X.

Before an expression is evaluated, it is parsed into a tree of subexpressions based on the usual sort of precedence rules found in most languages. Then it is evaluated by applying operators to objects according to two evaluation rules, one for ordinary operators and one for *special form* operators. To evaluate an expression,

1. If the operator is not a special form, first evaluate the operator's subexpressions (left-to-right), then apply the operator to the resulting objects;
2. If the operator is a special form, the operator is applied without prior evaluation of its subexpressions; the operator may cause any or all of its subexpressions to be evaluated.

The rules are applied recursively and must be augmented by two additional rules which are the basis of the recursion:

1. A constant (e.g., 1, 2.3D5, 'FOO') evaluates to itself.
2. A simple variable (e.g., a, arm, current_speed) evaluates to its binding (not a copy).

The details of the evaluation rule are only relevant when evaluation of a subexpression causes a *side effect*, i.e., when the value of some variable is changed during the course of evaluating the subexpression. We will consider several examples.

Example: <a, b> = <1, 2>
This expression assigns 1 to a and 2 to b. The left-hand-side evaluates to an aggregate of the bindings of a and b (not copies of the bindings); the right-hand-side evaluates to the aggregate <1,2>. The usual mapping rules then cause the assignment operator to be "distributed" over the corresponding aggregate elements.

There are some circumstances where such behavior is not desired. For those cases, AML/X has a *copy operator*, denoted by %. An expression of the form %expr evaluates to a *copy* of

```
diagonal:  SUBR ( m );
   ## Returns the diagonal elements of the n x n matrix m
   n:  NEW AGGSIZE ( m );   ## Number of rows (cols) in m
   i:  NEW 0;               ## Index over rows/cols
   RETURN ( WHILE ++i LE n DO COLLECT m(i, i) );
END;

id: NEW 3 OF 3 OF 0.;      ## Make a 3 by 3 matrix of zeros
diagonal(id) = 1.;         ## Set diagonal elements to 1
```

Figure 3. The subroutine diagonal returns the diagonal elements of a square matrix represented as a nested aggregate. Its use is illustrated by constructing the 3 by 3 identity matrix id.

the result of evaluating expr. Its use is illustrated in the following example.

Example: <a, b> = <b, a>
This expression might be written (incorrectly) to swap two variable values. Each side of the assignment evaluates to an aggregate of the bindings of a and b, but in different order. Applying the mapping rule, the expression is equivalent to <a = b, b = a>. Since these are evaluated left-to-right, the value of a will be "lost". The copy operator, used in the expression <a, b> = %<b, a>, causes a copy of the original values to be made before any of the assignments are done, thus achieving the desired effect.

Example: diagonal(m) = 1
The subroutine shown in Figure 3 uses WHILE..DO..COLLECT to construct an aggregate of the bindings of the diagonal elements of the matrix m. Since diagonal(a) evaluates to an aggregate of the bindings of the diagonal elements, diagonal(a) = 1 has the effect of setting all diagonal elements of a to 1. Again, the key point is that variables in AML/X evaluate to their bindings, *not* copies of those bindings, and the WHILE..DO..COLLECT operator aggregates but does not copy the successive values of the loop expression.

3.5 Subroutines

This section describes AML/X subroutines and variable declarations. A subroutine is defined by a statement of the form

```
subrname: SUBR(...formal_arguments...)
   declarations
   ...
   statements
END;
```

This statement really consists of two parts, the subroutine expression itself (SUBR...END) and the variable name subrname. The statement defines a subroutine and makes it the binding of the variable subrname. There may be any number of formal arguments, including zero. The *body* of the subroutine consists of any number (including zero) of *local variable declarations* followed by any number (including zero) AML/X statements.

A subroutine is called when a subroutine object is *applied* to an argument list, as follows:

```
subrexpr(actual_arg_1,...,actual_arg_n)
```

The subroutine to which the expression subrexpr evaluates is called with the specified actual arguments.

Local Variables and Declarations: Local variables are variables whose names are known only within the extent of a particular subroutine. A local variable is defined by being *declared* at the beginning of a subroutine. Variable declarations (except labels and internal subroutines) are of the form

```
varname: declarator init_expr;
```

where varname is the name of the variable being declared, declarator specifies various properties of the local variable, and init_expr is an arbitrary AML/X expression that defines the variable's type and initial value. The effect of a variable declaration is to associate a variable name with the storage that holds its binding, as defined by four attributes:

Memory space: Determines which memory space the binding is stored in. If the binding is in the stack, it will be discarded when the subroutine terminates.
Constant: Determines whether or not the value of the binding can be altered once it is initialized.
Copy: Determines whether the result or a copy of the result of evaluating the initialization expression becomes the binding.
Persistent: The binding is created when the subroutine is loaded and is reestablished each time the subroutine is invoked.

Possible declarators and their attributes are shown in F'gure 4.

A subroutine definition contained in another subroutine is an *internal subroutine* and can only be called from within the containing subroutine. Free variables in internal subroutines are bound lexically. Labels are declared by prefixing any statement by a name, as in

```
labname: stmt;
```

Internal subroutines and labels are two exceptions of the rule that declarations appear at the beginning of a subroutine. In this case, ease-of-use seemed to outweigh consistency.

Arguments: A simple formal argument is a variable name, implicitly declared as a BIND declaration, and bound to the corresponding actual argument when the subroutine is called. In effect, arguments are passed by reference. The caller can specify that an argument be passed by value by preceding the actual argument with the copy operator (%). The formal argument can also be preceded by the copy operator, in which case the argument is passed by value regardless of how the actual argument is passed. This is illustrated in the following example:

Declarator	Memory Space	Constant	Copy	Persistent
NEW	Stack	No	Yes	No
NEW CONSTANT	Stack	Yes	Yes	No
CONSTANT	Stack	Yes	Yes	No
STATIC	Heap	No	Yes	Yes
STATIC CONSTANT	Heap	Yes	Yes	Yes
BIND	—	—	No	No
STATIC BIND	—	—	No	Yes

Figure 4. Variable declarators

```
fact: SUBR(%i)
  ## Returns i factorial (i GE 1)
  f: NEW i;
  WHILE --i GT 1 DO f *= i;
  RETURN(f);
END;
```

AML/X provides a way to specify values for missing arguments. If the formal argument has the form

```
formal_arg DEFAULT expr
```

and the corresponding actual argument is supplied, then formal_arg is processed as described above. However, if the corresponding actual argument is missing, or if its value is DEFAULT, then expr is evaluated and its result becomes the binding of formal_arg. Thus in the code

```
s: SUBR(p, tol DEFAULT 1.0e-6)
  ...
END;
s(3);
s(3, 1.0e-8);
```

tol is bound to 1.0e-6 the first time s is called and to 1.0e-8 the second time.

Subroutines can also access excess actual arguments using the predefined variable ACTUAL_ARGS, which is bound to an aggregate of the actual arguments.

It is easy to write *generic subroutines* in AML/X because it is not necessary to declare the type of formal arguments. This is often convenient, especially for small programs, but can lead to programming errors and make it very difficult to compile efficient code for the subroutine. In keeping with our philosophy of letting the user make the trade-off between flexibility and efficiency, AML/X allows optional type declarations for formal arguments. A formal argument (possibly including a DEFAULT clause) can be followed by a type specification of the form

```
MUSTBE type_spec
```

where type_spec is an aggregate of types. If the corresponding actual argument is not one of the specified types, an exception is raised.

Exiting from Subroutines: Any AML/X object can be returned as the result of a subroutine call by passing the object to the RETURN built-in subroutine. The object returned is not copied unless it would be destroyed by termination of the subroutine (e.g., a NEW variable). Therefore a variable binding can be returned and a subroutine call can be used on the left-hand side of an assignment, as illustrated in Figure 3.

The built-in subroutine CLEANUP can be called while executing a subroutine to request an action when the subroutine terminates. For example, suppose a subroutine which opens a file should always close it, even if the subroutine terminates because of some error condition. This can be done by the code shown in Figure 5.

3.6 Exception Handling

When an error is detected during program execution an *exception* is *raised*. The action taken by a program when an exception is raised is determined by the *exception handler* defined for that particular exception. The design of AML/X's exception handling is described in [18].

Each possible exception has a name, which is represented by a SYMBOL. When an exception is raised, either by the system or by the user through the RAISE_EXCEPTION built-in subroutine, AML/X finds the most recent activation of a block containing a variable of that name which was declared as a HANDLER. If none is found and a variable of the appropriate name exists at top-level, it is used. The binding of that variable is the exception handler used.

The type of the binding determines what kind of action is taken, as follows:

EXPR: The EXPR is evaluated and the result becomes the result of the exception handler.

SUBROUTINE: The SUBROUTINE is called. The arguments passed provide the SUBROUTINE with detailed information about the exception that occurred. The result of the subroutine call becomes the result of the exception handler.

LABEL: The LABEL is branched to. The exception handler has no result.

BOOLEAN: The BOOLEAN object is set to TRUE and the result of the exception handler is TRUE.

SYMBOL: The value of the SYMBOL is used as the name of another exception to raise. This allows exceptions to be grouped hierarchically into exception groups, each consisting of several exceptions all handled by the same exception handler.

```
file_update: SUBR()
   c: NEW OPEN('file.name','w'); ## Open file
   CLEANUP( $(CLOSE(c)) );       ## Request cleanup action
   ...                           ## Processing code
END;
```

Figure 5. An example of a subroutine cleanup action: The expression `$(CLOSE(c))` passed to `CLEANUP` is an (unevaluated) expression object which will be evaluated when `file_update` terminates.

A program can use the `EXCP_BINDING` built-in subroutine to determine the current exception handler for a specified exception. In this way, a subroutine can decide to let an exception be handled by its caller if its caller has an appropriate handler, or can handle the exception itself if the caller doesn't provide a handler. For example, in the code in Figure 6, the subroutine `default_handle_set` is used in `foo` to bind `EXCP_ZERODIV` to the caller's handler if it exists, or to a boolean flag if it doesn't exist.

Most system-defined exceptions are *continuable*, meaning that execution continues from where the exception was raised and the result of the exception handler becomes the result of the operator that caused the exception. The operation is not "retried", although the exception handler is free to retry the operation or provide a reasonable result, as in

```
EXCP_ZERODIV: HANDLER SUBR()
  ## Return largest possible number
  RETURN(MAXVAL(LONG_REAL));
END;
```

User exceptions can be either continuable or non-continuable.

3.7 Object-oriented Programming

The use of abstraction is a very powerful tool for building large programs. Powerful or complicated abstractions can be implemented by using simpler ones so that each implementation is small and (presumably) easy to understand. AML/X supports abstraction by providing *classes*, a mechanism for defining new objects and operations on them. A class defines a new type in the language; a *class instance* is a particular object derived from a class. This is analogous to built-in types and instances of the built-in types: for example the number 2 is an instance of the type `INT`. Thus, if one writes a class definition for complex numbers, each instance of that class would correspond to a particular complex number. The data for each instance is held in its *instance variables*, which are accessible only from within the class unless access elsewhere is granted explicitly.

Class Definitions: A class definition is defined by a statement of the form

```
classname: CLASS(...formal_arguments...)
  IVARS
    instance variable declarations
  END;
  declarations
  initialization statements
  methods
END;
```

This statement defines a `TYPE` and makes it the binding of `classname`. Each declaration in the `IVARS` section defines an instance variable. All of the non-`STATIC` declarators shown in Figure 4 can be used. Figure 7 shows part of a simple class definition for vectors.

Class Instantiation: A class instance is created (the "class is instantiated") by calling the class definition as one would a subroutine, the only difference being that a class returns an instance containing the current bindings of the class' instance variables. Thus, `vector(1,2,3)` would return an instance of `vector` with instance variables `1E0`, `2E0`, and `3E0`.

Methods and Operator Overloading: Classes are only useful if there is a way to do something to class instances. A *method* is a special kind of internal subroutine that (1) is contained in a class definition but can be called from outside of the class definition, and (2) has access to the instance variables of an instance of the class. A method is invoked by executing an expression of the form

```
obj_exp.method_name(...formal_argumens...)
```

where `obj_exp` is an expression that evaluates to a class instance and `method_name` is the name of a method. The method executes exactly like an ordinary subroutine except that the instance variables are bound to the values of the instance variables contained in the instance instead of to the result of evaluating their initialization expressions. Also, the predefined variable `SELF` is bound to the class instance itself.

AML/X operators can be extended to class instances, or *overloaded*, on a class-by-class basis by associating a method with the operator. This is done simply by having in the class a method whose "name" is a literal form of the operator to be overloaded. If the left operand of an operator is a class instance, the corresponding operator method in the appropriate class definition is invoked with the right operand as actual argument; if the left operand is an instance of a built-in type but the right operand is a class instance, the corresponding *modifier method* is invoked passing the left operand as actual argument. This allows non-commutative operators to be overloaded.

Exposed Instance Variables: Ordinarily, instance variables are not accessible except within the class definition or through method calls. Direct access to instance variables can be explicitly granted by declaring them to be `EXPOSED` as in

```
IVARS
  x: EXPOSED NEW REAL();
END;
```

An exposed instance variable is referenced by an expression of the form:

```
instance.inst_var_name
```

Note that a class definition containing only exposed instance variables is equivalent to C's structures and PASCAL's records.

Exposed instance variables were added to the language in response to user's complaints that they often had to write a method just to access a single instance variable. However, because they expose the data representation used by a class, they

```
default_handle_set: SUBR(ex_name, new_handler)
  ## If the caller of the caller of this routine does not
  ## have an exception handler for the exception named
  ## ex_name, return the specified new handler; otherwise,
  ## return the existing handler.

  existing_handler: BIND EXCP_BINDING(ex_name, CALLER(CALLER()));

  RETURN( IF existing_handler NE UNBOUND THEN existing_handler
                                         ELSE new_handler      );
END;

foo: SUBR()
    ## Set up handler for EXCP_ZERODIV
    EXCP_ZERODIV: HANDLER BIND
       default_handle_set($EXCP_ZERODIV, FALSE);
    ...
END;
```

Figure 6. Providing a default exception handler: The subroutine foo binds the result of calling default_handle_set to EXCP_ZERODIV, thus making it the exception handler for dividing by zero. The subroutine default_handle_set first determines the exception handler binding in its caller's caller (i.e., foo's caller). If there is one, it is returned for use; otherwise the new_handler is returned for use as the exception handler.

violate the abstraction that classes were intended to provide. PRIVATE EXPOSED instance variables, which only allow direct access to instance variables from within the class definition, were introduced so that data representation could be exposed inside the class definition but remain hidden from outside view.

4.0 Examples

4.1 Cartesian Data Types

Data types for vectors, rotations, and coordinate transformations are often provided in special-purpose languages for robotics and CAD [e.g., 14, 15, 19, 20]. The conciseness and consistency checking provided by such types, compared to the subroutine libraries providing comparable functions for general purpose languages, significantly improves programmer productivity and program readability. Unfortunately, users of special-purpose languages are often stuck with whatever internal representation and function set the system implementers have chosen to provide.

This section illustrates the use of AML/X classes and operator overloading to implement these data types in a way that provides both expressive power and easy customizing.

Vectors: Vectors are represented by three real numbers, stored in EXPOSED instance variables x, y, z. Methods overloading the normal arithmetic operators can be provided for vector addition, subtraction, and scaling. "Multiplication" of two vectors is used for vector inner product and "exponentiation" is used for cross product. A method for assignment can also be provided. A sketch of such a class definition is given and several standard constant vectors are declared in Figure 7.

Rotations: Rotations are commonly represented as 3x3 orthogonal matrices. This representation is easily understood and is computationally efficient if many vectors are to be rotated. On the other hand, it is wasteful of storage, subject to numerical inconsistencies, and computationally expensive for many operations, including composition and specification from angles.

As an alternative, we have sketched in Figure 8 a class definition for rotations that uses quaternions[21, 22] as the underlying representation. In this case, EXPOSED PRIVATE instance variables are used in order to hide implementation details while permitting efficient execution within methods. Two "class" methods,

```
r = rotation.polar(axis_vector, angle);
r = rotation.euler(abt_z_1,abt_y,abt_z_2);
```

permit rotations to be specified either as a right-handed twist about a specified axis or as a sequence of rotations about cardinal axes. Multiplication is overloaded to provide for composition of two rotations and rotation of a vector and division is overloaded to provide for formation of an inverse rotation and for multiplication by the inverse.

Two methods,

```
<axis_vector,angle> = r.polar_parms();
<abt_z_1,abt_y,abt_z_2> = r.euler_parms();
```

invert the polar and euler methods, respectively. One potential problem with the latter method arises when the second angle, corresponding to rotation about the "y" axis, is zero. In this case, only the sum of the first and third angles is determined. By default, the third value will be set to zero and an exception, EXCP_degen_rot, is raised. However, the exception handler can override the default. For example, the simplified kinematic solution procedure shown in Figure 9 uses an exception handler to divide the angle sum evenly between the first and third wrist joints.

Transformations: Arbitrary coordinate transformations, consisting of rotation followed by translation, are straightforwardly implemented using the class definitions for vectors and rotations. In a typical class definition (not shown), multiplication would be overloaded to provide transformation of vectors and composition, and division would be overloaded to provide inverses and composition with inverses. A class method for co-

```
vector: CLASS(xx DEFAULT 0.0, yy default 0.0, zz DEFAULT 0.0)

  IVARS
    x: EXPOSED NEW REAL(xx);
    y: EXPOSED NEW REAL(yy);
    z: EXPOSED NEW REAL(zz);
  END;

  $*: METHOD(v)        ## Inner product and scaling
    SELECT (?v)
      CASE vector THEN RETURN(x*v.x+y*v.y+z*v.z)
      OTHERWISE RETURN(vector(x*v, y*v, z*v))
    END;
  END;
  $*: MOD_METH(s) RETURN(vector(s*x, s*y, s*z)); END;

  $**: METHOD(v MUSTBE <vector>)  ## Cross product
    RETURN(vector(y*v.z-z*v.y, z*v.x-x*v.z, x*v.y-y*v.x));
  END;

  ...

  $=: METHOD(v)
    <x, y, z> = SELECT (?v)
                  CASE vector THEN <v.x, v.y, v.z>
                  OTHERWISE v
                  END;
    RETURN(SELF);
  END;

  uvect: METHOD() RETURN(SELF/sqrt(self*self)); END;
END;

null_vector: STATIC CONSTANT vector(0, 0, 0);
x_axis:      STATIC CONSTANT vector(1, 0, 0);
y_axis:      STATIC CONSTANT vector(0, 1, 0);
z_axis:      STATIC CONSTANT vector(0, 0, 1);
```

Figure 7. Class definition for vectors: Methods that overload the addition, subtraction, and division operators have been omitted for brevity.

ercing vectors and rotations to transformations would also be useful.

4.2 Coordinate Frames and Affixment

Coordinate transformations arise naturally from part-subpart relationships in both robot and CAD programming. If the location (i.e., position and orientation) of Part A relative to the workstation is given by a transformation, `frame_a`, and `trans_ab` gives the location of a Subpart B relative to A, then the location of B relative to the workstation is given by `frame_a*trans_ab`. Similarly, if the location of C relative to B is given by `trans_bc`, then the location of C relative to the workstation is given by `frame_a*trans_ab*trans_bc`. In practice, these expressions become very cumbersome and tend to interfere with the readability of programs. To get around this, AL [14] introduced the concept of *affixment*, in which part-subpart relationships and similar dependencies were declared explicitly, as in

```
AFFIX part_b TO part_a AT trans_ab;
```

Programs then simply referred to `part_b` to get the current location of Part B. If Part A was moved or if a new value for its location was determined by sensing, then the location value for Part B was updated automatically.

One of the interesting aspects of the AL implementation of affixment [23] was that recomputation of coordinate frame values was deferred until they were needed, but the values were saved to eliminate needless recomputation. This saving can be quite important in robotic applications where parts are being moved about the workstation and where the expressions involved in recomputation may involve a long chain of affixments. An AML/X implementation of much the same idea is shown in Figure 10. Once again, classes and operator overloading are used to provide a new data type, `frame`, whose value corresponds to a coordinate system.

Figure 11 illustrates the use of this data type in a simple assembly application. Figure 11(a) shows a box and cover plate being delivered to an assembly station on a small tray. The coordinates of the box and cover are initially known relative to the tray. Furthermore, grasping points relative to the box and cover have been defined. The problem is to use vision to locate the tray, then use a vision routine to locate the box and cover more precisely, based on the tray location. Finally, place the cover on the box and pick up the box. A program to accomplish this is sketched in Figure 11(b).

```
rotation: CLASS(ss DEFAULT 0.0, vv DEFAULT null_vector MUSTBE <vector>)

  IVARS
    s: PRIVATE EXPOSED NEW REAL(ss);
    v: PRIVATE EXPOSED NEW vv;
  END;

  $=: METHOD(r) <s,v> = <r.s, r.v>; END;

  polar: CLASS_METH(axis MUSTBE <vector>, angle)
    RETURN(rotation(cos(angle/2), sin(angle/2)*axis.uvect()));
  END;

  euler: CLASS_METH(a, b, c)
    RETURN(rotation.polar(z_axis,a) * rotation.polar(y_axis,b) * rotation.polar(z_axis,c));
  END;

  $*: METHOD(p MUSTBE <rotation,vector>)
    SELECT (?p)
      CASE rotation THEN RETURN(rotation(s*p.s-v*p.v, p.s*v+s*p.v+v**p.v))
      CASE vector THEN RETURN((SELF*rotation(0,p)/SELF).v)
    END;
  END;

  $/: METHOD(p MUSTBE <rotation>)
    RETURN(rotation(s*p.s+v*p.v, p.s*v-s*p.v-v**p.v))
  END;

  $/: MOD_METH(p)
    IF p NE 1 THEN RAISE_EXCP($EXCP_invalid_inv,,<p,SELF>,TRUE);
    RETURN(rotation(s,-v));
  END;

  euler_parms: METHOD()
    ad:   BIND s**2 + v.z**2;
    bc:   BIND v.x**2 + v.y**2;
    beta: BIND acos((ad-bc)/(ad+bc));
    gpa:  BIND atan2(v.z, s);
    gma:  BIND IF beta NE 0 THEN atan2(v.x, v.y)
           ELSE BEGIN rslt: BIND RAISE_EXCP($EXCP_degen_rot,, gpa, TRUE);
                  IF is_3_reals(rslt) THEN RETURN(rslt) ELSE 0.0;
                END;  ## See also Figure 9
    RETURN(<gpa-gma, beta, gpa+gma>);
  END;

  polar_parms: METHOD()
    s_sqd: BIND v**2;
    RETURN(IF s_sqd EQ 0.0 THEN <z_axis, 0.0>
                           ELSE <v/sqrt(s_sqd), acos(s**2-s_sqd)>);
  END;

  is_3_reals: SUBR(val)
    RETURN( ISAGG(val) CAND AGGSIZE(VAL) EQ 3 CAND ALL(?VAL EQ REAL) );
  END;

END;

EXCP_degen_rot: BIND HANDLER FALSE;  ## Default: ignore exception
null_rot: STATIC CONSTANT rotation.polar(z_axis, 0);
```

Figure 8. Class definition for rotations

The declarations create an "affixment tree" of frames. The instance variables associated with each frame specify the parent frame, the offset of the frame relative to its parent, the present value of the frame—i.e., its transformation relative to the workstation—a "mark" counter used to determine whether the value is valid, and a flag specifying whether the affixment is "rigid". The mark counter is incremented every time a new value is saved, and a frame's value is valid if and only if its mark counter is greater than its parent's. "Rigid" affixments are those

```
hand_vector: STATIC CONSTANT vector(0,0,9);

solve_arm: SUBR( f MUSTBE <frame> )
  t: BIND f.xf();  ## Frame transformation
  cj: BIND t.v - t.r * hand_vector;
  RETURN( <cj.x, cj.y, cj.z> # (t.r).euler_parms() );

  EXCP_degen_rot: HANDLER SUBR(a, b, ang_sum);
     RETURN(<ang_sum, 0., ang_sum>/2);
  END;
END;
```

Figure 9. Simplified kinematic solution procedure for a cartesian robot such as the IBM 7535: This subroutine returns an aggregate of six real numbers giving joint values corresponding to specified hand frame. Note the use of an exception HANDLER subroutine to override the default handling of degenerate wrist rotations.

in which updates to the frame value are to cause the parent's value to be updated as well.

The initial assignment statement causes the value of tray to be updated, and its mark to be incremented. The second statement first causes the value of cover to be computed as part of the call to locate_object. Since tray has been updated, its mark counter is higher than that of cover, so the value is obtained by obtaining a valid value for tray and then composing the result with the offset stored for cover. The assignment then updates the value stored in cover a second time. The third statement repeats the process for box.

Subsequent statements call subroutines to pick up the cover, place it on the box, etc. The fragments from grasp_object and move_object illustrate the use of affixment to simplify programming. grasp_object moves the robot to the specified grasping point, closes the gripper, and then affixes the object (by assumption, the most remote rigidly affixed ancestor) to the robot. Subsequent motions of the robot will cause all location attributes of the object to be updated. move_object verifies that motion_frame is affixed to the robot and then moves the robot so that the value of motion_frame is equal to destination.

5.0 Implementation

AML/X is implemented by a portable interpreter written in C. It runs on the IBM 370 family of machines under VM/CMS, on the IBM PC under DOS and XENIX, on the IBM RT/PC under AIX, and on several other machines. Facilities exist on all machines for writing C subroutines callable from AML/X; on the VM/CMS implementation, there are also interfaces to Fortran and PL/1.

6.0 Experience and Conclusions

AML/X has now been in use for about a year in our research in robotics, computer-aided design, and machine vision. Execution speed of the interpreter has proved adequate for robotics applications. As anticipated, however, the interpreter is too slow for production use in the lowest layers of more complex systems, especially in CAD applications. We have begun work on a prototype compiler that should resolve this issue. Meanwhile, several of our researchers find AML/X sufficiently expressive that they prototype low level geometric data structures and algorithms in AML/X and recode in C where necessary for efficiency. Classes, including operator overloading, are often used in this work, and the resulting programs are both readable and modifiable.

AML/X has also been used as a programming "front-end" to a powerful geometric modelling system [17]. In this case, class definitions for geometric objects have been written in AML/X but the actual data representation is created and manipulated by the modelling system. Each class instance essentially contains a "handle" on the data maintained by the modelling system. Methods implement geometric operations by passing these "handles" to the modelling system, which then does the necessary computation. Our initial (limited) experience with this use of AML/X in the higher layers of a system is that it provides a very powerful programming environment with very reasonable performance. The drawing in Figure 11 was produced by an AML/X program running on this system.

The need for concurrency can to some extent be met by simple interfaces to operating system services. However, in response to the requirements of automation programming, we have begun to consider providing concurrency directly in the language.

Providing interfaces to other languages has allowed us quick access to a variety of existing code, ranging from mathematical subroutines, to graphics routines, to a large modelling system. We expect that AML/X will continue to be used at the highest layers of large systems built from existing components and that using it in this way will help us to integrate various automation technologies.

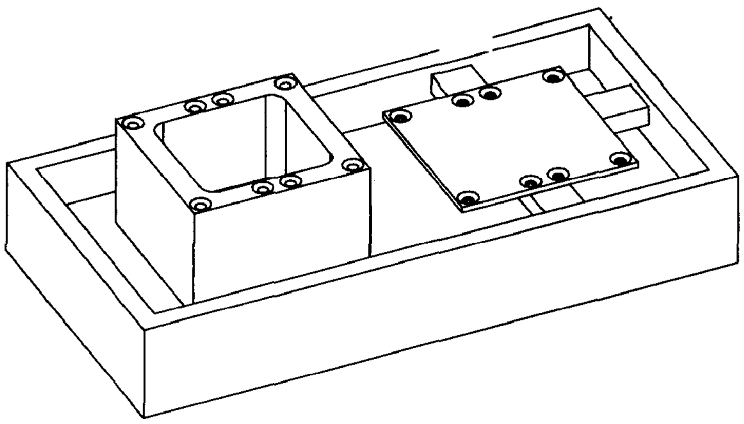
Acknowledgments

AML/X is a descendant of AML (A Manufacturing Language) and owes much to its developers. As early users of AML/X, Georg Maier and Vijay Srinivasan at IBM Yorktown provided especially valuable suggestions on ways to improve the language. Dave Klein of the Manufacturing Systems Program (MSP) at IBM Boca Raton designed and implemented the garbage collector. Ken Morgan, also of MSP, implemented most of the debugging facilities. Many others in the Automation Research group at IBM Yorktown and the Manufacturing Systems Program at IBM Boca Raton have also made useful comments and suggestions.

```
frame: CLASS(afx  DEFAULT NULL MUSTBE <REF,frame>,
             ofst DEFAULT trans(),
             rigidly DEFAULT FALSE MUSTBE <BOOLEAN> KEY)

  frame_counter: STATIC LONG_INT(0);

  IVARS
    parent: PRIVATE EXPOSED NEW IF ?afx EQ REF THEN afx ELSE &afx;
    offset: PRIVATE EXPOSED NEW trans.coerce(ofst);
    value:  PRIVATE EXPOSED NEW IF parent EQ NULL THEN
                                trans() ELSE (!parent).xf()*offset;
    mark:   PRIVATE EXPOSED NEW frame_counter++;
    rigid:  PRIVATE EXPOSED NEW BOOLEAN(rigidly);
  END;

  $=: METHOD(f)
    value = trans.coerce(f);
    IF parent NE NULL THEN
       IF rigid THEN (!parent) = value/offset
       ELSE offset = (1/(!parent).xf())*value;
    mark = frame_counter++;
    RETURN(%value);
  END;

  $*: METHOD(b) RETURN(SELF.xf() * b); END;

  $/: METHOD(b) RETURN(SELF.xf() / b); END;

  xf: METHOD() SELF.validate(); RETURN(%value); END;

  unfix: METHOD() parent = NULL; rigid = FALSE; END;

  affix_to: METHOD(afx MUSTBE <REF,frame>, ofst,
                   rigidly DEFAULT FALSE MUSTBE <BOOLEAN> KEY)
    IF ?ofst EQ DEFAULT THEN SELF.validate();
    SELF.unfix();
    rigid = rigidly;
    parent = IF ?afx EQ frame THEN &afx ELSE afx;
    IF ?ofst EQ DEFAULT THEN
      offset = (1/((!parent).xf()))*value
    ELSE
      BEGIN mark = 0; offset = trans.coerce(ofst); END;
  END;

  validate: PRIVATE METHOD()
    if parent EQ NULL then RETURN();
    (!parent).validate();
    IF mark LE (!parent).mark THEN
       BEGIN value = (!parent).value*offset; mark = frame_counter++; END;
  END;

  rigid_ancestor: METHOD()
    RETURN(IF rigid CAND parent NE NULL THEN (!parent).rigid_ancestor() ELSE SELF);
  END;

  has_ancestor: METHOD(f)
    RETURN(IF SELF EQ f THEN TRUE
           ELSE IF parent EQ NULL THEN FALSE
           ELSE (!parent).has_ancestor(f));
  END;

END;
```

Figure 10. Class definition for Cartesian frames and affixments

(a)



(b)

```
tray:        NEW frame();
cover:       NEW frame(tray,  trans(...));
cov_grasp:   NEW frame(cover, trans(...), rigidly=TRUE);
box:         NEW frame(tray,  trans(...));
box_top:     NEW frame(box,   trans(...));
box_grasp:   NEW frame(box,   trans(...), rigidly=TRUE);

tray =  locate_object(DEFAULT, ...); ## no a priori info
cover = locate_object(cover,   ...); ## locate cover better
box =   locate_object(box,     ...); ## locate box better

grasp_object( cover_grasp, ...);     ## grasp the cover
move_object( cover, box_top, ... );  ## move it
release_object( ... );               ## let go
grasp_object( box_grasp, ... );
move_object( ... );

...

grasp_object: SUBR(grasp_frame, ... );
   ...
   move_robot(grasp_frame, ...);
   close_gripper( ...);
   (grasp_frame.rigid_ancestor()).affix_to(robot);
   ...
   END;

move_object: SUBR(motion_frame, destination, ...);
   ...
   IF NOT motion_frame.has_ancestor(robot) THEN
      RAISE_EXCP( ... );
   move_robot(destination/motion_frame*robot, ... );
   ...
   END;
```

Figure 11. Simple robotic assembly task: (a) Initial situation and (b) Sketch of program. The models in (a) were implemented using an AML/X front-end to the IBM Geometric Design Program [17].

References

[1] D.D. Grossman, "Programming a computer controlled manipulator by guiding through the motions," IBM Research Report RC6393, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1977.

[2] R.H. Taylor, P.D. Summers, and J.M. Meyer, "AML: A Manufacturing Language," *Intl. J. Robotics Research*, vol. 1, no. 3, pp. 19-41, Fall 1982.

[3] R. H. Taylor and D. D. Grossman, "An Integrated Robot System Architecture", *IEEE Proceedings*, vol. 71, pp. 842-855, July 1983.

[4] M.A. Lavin and L.I. Lieberman, "AML/V: An Industrial Machine Vision Programming System," *Intl. J. Robotics Research*, vol. 1, no. 3, pp. 42-56, Fall 1982.

[5] J. Korein, G. Maier, R. Taylor and L. Durfee, "A Configurable System for Automation Programming and Control," *Proc. 1986 IEEE Conf. on Robotics and Automation*, San Francisco, pp. 1871-1877, April 1986.

[6] L.R. Nackman, M.A. Lavin, R.H. Taylor, and W.C. Dietrich, Jr., "AML/X User's Manual," IBM Research Report RA 175, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, (1986).

[7] C.A.R. Hoare, "Hints on Programming Language Design," Keynote address given at the ACM SIGACT/SIGPLAN Conf. on Prinicples of Programming Languages, Boston, (1973) as quoted on pp. 255 and 257 of C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, New York: John Wiley, 1982.

[8] M.A. Wesley, "Construction and Use of Geometric Models," in *Computer Aided Design*, J. Encarnacao, ed., Lecture Notes in Computer Science 89, Springer Verlag, 1980.

[9] T. Lozano-Perez, "Robot Programming," *Proc. of the IEEE*, vol. 71, no. 7, pp. 821-841, July 1983.

[10] M.A. Lavin and L.I. Lieberman, "AVL0 -- A Vision Language," IBM Research Report 8390, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, (1980).

[11] C. Eastman and M. Henrion, "Glide: A Language for Design Information Systems," Proc. ACM SIGGRAPH'77, *Computer Graphics*, vol. 11, no. 2, pp. 24-33, Summer 1977.

[12] P. Will and D. Grossman, "An experimental system for computer controlled mechanical assembly", *IEEE Trans. Comput.*, vol. C-24, p. 879., 1975.

[13] R. Evans, et. al., "Software system for a computer controller manipulator", IBM Res., Yorktown Heights, NY, Rep. RC-6210, 1977.

[14] R. Finkel, R. Taylor, R. Bolles, R. Paul and J. Feldman, "AL, A Programming Language for Automation," Stanford Artificial Intelligence Laboratory Memo AIM-243, Stanford University, 1974.

[15] V. Hayward and R. Paul, "Robot Manipulator Control under UNIX," from TR-EE 84-10, Purdue University School of Electrical Engineering, pp. 22-34, Jan. 1984.

[16] A. Bloch, *Murphy's Law and other reasons why things go gnorw.* Los Angeles: Price/Stern/Sloan, 1977.

[17] M.A. Wesley, T. Lozano-Perez, L.I. Lieberman, M.A. Lavin, and D.D. Grossman, "A Geometric Modeling System for Automated Mechanical Assembly," *IBM J. Res. Dev.*, vol. 24, pp. 64-74, Jan. 1980.

[18] L.R. Nackman and R.H. Taylor, "A Hierarchical Exception Handler Binding Mechanism," *Software--Practice and Experience*, vol. 14, no. 10, pp. 999-1003, Oct. 1984.

[19] B. Shimano, "VAL: An industrial robot programming and control system", *Proc IRIA Sem. of Languages and Methods of Programming*, Rocquencourt, France, pp. 47-59, June 1979.

[20] C.M. Brown, "PADL-2: A Technical Summary," *IEEE Comp. Graphics & Applications*, vol. 2, no. 2, pp. 69-84, Mar. 1982.

[21] W. R. Hamilton, *Elements of Quaternions*, Third Edition, New York: Chelsea Pub. Co., 1969.

[22] R. H. Taylor, "Planning and execution of straight line manipulator trajectories", *IBM J. of R. & D.*,vol. 23, no. 4, pp. 424-436, July 1979.

[23] R. H. Taylor, *A Synthesis of Manipulator Control Programs from Task-Level Specifications.*, PhD Dissertation, Memo AIM-282, Artificial Intelligence Laboratory, Stanford Univ., Stanford, CA, 1976.