TECHNICAL PAPER

MS84-493

Robots Are (Nothing More Than) Abstract Data Types

abstract

During the past decade, data and program abstractions have emerged as a major organizational concept in programming languages. They offer particular advantages for large programming problems. Robot programming is no exception. These concepts can be used to program a robot-based manufacturing cell. Using an experimental robot-based manufacturing cell as an example, a strategy for robot programming based on Ada is described. The central features of Ada, data abstraction and program abstraction through generics and operator overloading are illustrated. The principal advantages and difficulties of data and program abstractions as they may be realized through ADA are summarized. (Ada is a registered trademark of the Department of Defense.)

authors

R.A. VOLZ
Professor
University of Michigan
Ann Arbor, Michigan

T.N. MUDGE
Associate Professor
University of Michigan

conference

Robotics Research: The Next Five Years and Beyond
August 14-16, 1984
Bethlehem, Pennsylvania

index terms

Robots
Robotics
Automation
Programming

1984

Society of Manufacturing Engineers • One SME Drive • P.O. Box 930
Dearborn, Michigan 48121 • Phone (313) 271-1500

ABSTRACT

During the past decade, data and program abstractions have emerged as a major organizational concept in programming languages. They offer particular advantages for large programming problems. Robot programming is no exception. This paper describes how these concepts can be used to program a robot-based manufacturing cell. Using an experimental robot-based manufacturing cell as an example, a strategy for robot programming based on Ada[1] is described. The central features of Ada, data abstraction and program abstraction through generics and operator overloading are illustrated. The principal advantages and difficulties of data and program abstractions as they may be realized through Ada are summarized.

I. INTRODUCTION

With the event of robot-based manufacturing cells, the need for an implementation language with modern software tools to program these cells has grown in importance. Unfortunately, the current high level languages used to implement the real-time requirements of manufacturing systems lack critical language tools, such as data abstraction, that facilitate programming in the large. Indeed, even the most sophisticated robot, numerically controlled (NC) tool, and related "manufacturing systems" programming languages presently in commercial use support neither data abstraction nor other features appropriate for large scale programming [Shi82, GSC82, VMG84]. They are, therefore, unsuitable at the cell integration level.

This paper discusses the use of data and program abstractions for programming sensor-based robot systems. It is shown that abstractions provide a natural method of expression for robot and sensor operations and that is is possible to

[1]Ada is a registered trademark of the Department of Defense.

develop a problem oriented syntax through their use. The concepts are illustrated through the programming of an experimental robot cell consisting of a robot, a TV camera and a link to an off-line Computer Aided Design (CAD) database from which grip position and vision training information are obtained for the parts to be manipulated. The system is programmed in the new DoD language Ada [DoD83].

Ada was originally developed at the instigation of the DoD for programming embedded systems. Examples of embedded systems are, to quite one of the designers of the language, "those for process control, missile guidance or even the sequencing of a dishwasher" ([Bar84] p.xi). We would add to that list robot-based manufacturing cells. Ada is based on Pascal. However, significant extensions make it the first practical language to bring together important features that include data abstraction, separate compilation, multitasking, exception handling, encapsulation, and program abstraction through generics and operator overloading. These extension make Ada particularly appealing for programming large scale real-time embedded systems--a situation characteristic of robot-based manufacturing cells. Through fully validated compilers have only recently become commercially available, DoD's strong support of the language guarantees a large scale presence in the future.

II. DATA AND PROGRAM ABSTRACTIONS

During the past decade, data and program abstractions have emerged as a major organizational concept in programming languages [Sha80]. Abstract data types are defined in terms of the operations on the data rather than the storage structure of the data and the implementation of operations, both of which are hidden. Generally the abstract data type consists of a program unit that has two major parts: 1) a section that contains the names visible outside the program unit i.e., the name, type and header of all operations permitted to use the representation of the type, and 2) a section that contains the actual representation of the data type in terms of built-in or other defined data types together with the implementation of the operations allowed on the data. This second section is neither visible nor accessible outside of the program unit. The only ways a user may access the data are by the operations defined on it.

This data hiding provides a number of advantages important to large scale programming, including:

- hiding implementation details irrelevant to the application
- localization of implementation, making changes easier to perform
- reduction in the likelihood of accidental changes to the data by unrelated program segments
- improvement in the readability of the code.

These advantages are important to any large programming problem. Robot programming, particularly the programming of the complex sensor-based robots of the future, is no exception.

Expressing the motions of robot systems requires a number of high level data structures such as homogeneous transformations, joint coordinates, video images, image features, physical object parameters, tactile images, tactile image features, etc. The representations of many of these data structures are often similar and, if unrestricted access to them is allowed, all of the problems abstract data types were designed to protect against can easily occur. The arguments for using abstract data types for these structures is the same as in any large program.

Robots, sensors and other peripherals may be also viewed as abstract data types. Indeed, this is the natural way to view them. Typically, devices are attached to a computer through a rather narrow interface with a well defined protocol for communication between the computer and the device. A "device driver" is then written to present a usable interface to the user. The device driver is similar to the concept of an abstract data type in that it provides a visible interface to a device which cannot be accessed directly. However, device drivers usually act more as information transferring agents than as operations on the device. With a reorientation of the notion of a device driver, perhaps through a layer of software on top of it, this software interface can appear as a set of operations on the device, i.e., the device can be viewed as an abstract data type.

Program units are the basis for another level of abstraction. With program abstraction operations are defined on pseudo-objects rather than real objects. Then when the operations are needed for a real object a copy of the abstracted program is "instantiated" with the pseudo-object replaced by a real object. In this way, if the same operation is required for several different object types, different real instances of the program may be created by instantiating the abstracted program for each of the different types. It is less apparent where program abstraction fits in the context of robot programming. However, it has been studied both with respect to representation for different robots and with respect to a vision system for robots, and the concept will be illustrated later.

Data and program abstractions also can provide a basis for both tailoring the language to a problem oriented syntax and extending it to include new sensors, control functions or even access to CAD databases. The high degree of modularity accompanying the use of abstractions also suggests that it should be easier to extend the system to include different robots or port the software onto a new computer system.

Several experimental programming languages have been implemented that were designed containing features to support abstract data types, but, with the possible exception of Modula-2 [Wir82] and Concurrent Pascal [Bir77], Ada is the first that is likely to see wide-spread use. These concepts are thus being tested through use of Ada.

III. ADA FOR CELL PROGRAMMING

Ada has been expressly designed to facilitate the development and maintenance of large software systems through partitioning. Separate compilation allows a team of cell designers to work concurrently on the development of

separate subsystems. It also allows subsystems to be easily modified without affecting the rest of the system--an important feature for maintaining the system. Ada relies on data and program abstraction to simplify the construction of subsystem interfaces. Further, Ada provides multitasking and timing constructs, essential ingredients in manufacturing cells where there are typically several computation tasks that need to be performed in real-time.

In this section we describe in a general way the important features of Ada and a strategy that can be used for developing a robot and sensor programming system based on the use of Ada. The strategy and example described are based on the experimental system developed at the University of Michigan. The system is being used as an experimental vehicle for testing algorithms which link CAD to robot and sensor operations and for testing object-based architectures (see below) for robot/sensor systems. As of this writing, a vision system, a robot system, and algorithms for CAD-based vision training and determination of grip positions are functional. (The example below will illustrate some of these features while indicating the use of Ada.)

A. FEATURES OF ADA

The underlying philosophy of Ada is centered upon the use of *objects* for program design. An object is a data structure[2] having a unique identifier and an associated set of functions and procedures that can operate on it [Org82]. These "operators" are the only allowed means of manipulating the object. A number of advantages follow from this "object-based" programming methodology. Objects and their associated functions and procedures form natural boundaries along which to subdivide systems. In addition, because the structure of a data type is hidden from all but its associated operators, changes to the structure have a limited impact greatly simplifying program modification and maintenance. Thus, object-based programming provides a way to implement data abstraction.

Ada provides a construct called a *package* that allows the programmer to encapsulate objects and their associated functions and procedures. In addition, it has *private* types and *limited private* types that further restrict encapsulation so that objects thus typed, while visible to program parts, can only be manipulated through procedure and function references. Together these features permit the programmer to hide data structure implementation and create abstract data types. The package definition consists of two parts, a *specification* part and a *body*. The specification part introduces the data types, variables and procedures visible to the user of the package. The body contains the implementation of the package and may be accessed only by the mechanism stated in the package specification.

Program abstraction is possible through the use of *generic* packages and subprograms, and operator overloading. These are a step in the direction of polymorphic function implementation [Mil78, GoY80], and they allow, among other things, operations to be defined over a set of data types, thus providing a broader use of objects. An example is given in a later section. As shall be

[2]The meaning of the term "object" is not universally agreed upon; our usage is fairly narrow. See [Ren82] for a discussion of various viewpoints.

discussed further, the object and package concepts address the management and portability of complex software.

Ada also provides a *task* construct which is a means of dividing a program into logically concurrent operations with possible synchronization between them [REM81]. In addition to forming the basis for real-time operations, tasks also provide a means of increasing processing efficiency in a multiple processor environment. Syntactically, tasks bear a resemblance to packages in the sense that they both have a specification part and a body. However, the specification part of a task is used solely to declare the synchronization points or *entry* points to the task--the entry points indicate where messages are received/transmitted by a task.

The program and data abstraction capabilities of Ada can give rise to a large number of procedure calls which in turn add processing overhead to the program. Ada provides an *inline pragma* (a compiler directive which expands the subroutine source code in-line wherever called), thus eliminating some of the entry/exit overhead associated with procedure calls. The effectiveness of the pragma has not been widely tested as yet.

B. A STRATEGY FOR USING ADA

Manufacturing cell programming or robot programming with Ada depends upon three central ideas:

1. The use of Ada's data and program abstraction.
2. The use of Ada's extensibility.
3. The use of Ada's real-time multitasking capabilities.

The use of Ada for programming manufacturing cells begins with the definition of *objects* for the various physical and logical components in the cell and the interfaces to these objects. Among these are the problem oriented primatives one would like to have in a robot language. These objects are embedded in Ada packages. Various mechanisms can be easily implemented to (nearly) automatically make these objects available to the programmer. The robot programmer can then use these objects and interfaces as though they were part of the language specification.

To provide a concrete illustration of the use of Ada as the system implementation language for cell programming, a simplified version of the system mentioned above, consisting of a robot, a vision sensor and a link to a CAD database, is considered. The example also illustrates a related activity which we believe will be of great importance in the future and which impinges on the current paper: the use of CAD information for driving cell operations. Our goal is for the cell to be able to manufacture any part within some reasonable class without human intervention. In other words, the cell will be able to automatically adapt to a particular type of part once the part has been identified by the vision system. In addition, in contrast to present practice, there will be no loss of production time due to training either the robot or machine vision system [Hil80, Gle79, Per81]. Information to allow the vision system to identify a part and to allow the rest of the cell to perform the appropriate operations on the part is derived

from a CAD database. Therefore, it is a prerequisite that all the parts to be handled by the cell have been designed using a CAD system.

Four basic object types are defined in this illustration:

- ROBOT -- Provides the basic robot interface.
- POSITION -- Provides a set of data abstractions for part and robot locations.
- CAD_MODEL -- Provides CAD database access.
- VISION -- Provides an interface to the vision subsystem.

Each of these basic objects is associated with an Ada package. A view of a portion of the main programs and the specification for each of these basic objects gives an introduction to the object oriented design approach.

Figure 1 shows a portion of a sample main program. The action part of the example shown is to find a part on an input conveyor, move the robot to it and pick it up. It is assumed that the set of parts that could potentially appear on the conveyor are assigned to the variable SET_OF_PARTS. As mentioned earlier, the cell automatically adapts itself to handle any one of the set once it has been identified by the vision subsystem. The names of the parts are assigned to SET_OF_PARTS from a terminal. This information is transmitted to procedure MAIN by a command interpreter (not shown). Details such as following a particular speed profile or the handling of exceptions are omitted since they would tend to obscure the example. It is assumed that the geometry of the part is available in a CAD database and that off-line utilities are available to provide recognition information to the vision system (see next section) and the location on the part where it can be picked up (the grasp points are defined in a local coordinate system of the part itself)[WVW84].

The first part of this example identifies Ada packages which provide data types and services to the main program. The *with* and *use* clauses are the mechanism by which the robot environment is made available. (In the program parts shown, words in lower case bold are Ada key words. The upper case words are user-defined, or predefined, package, function, procedure, type or variable names.) The *with* clause tells the compiler that the programmer intends to use data types, procedures, and functions defined in the package named after the *with*. The *use* clause tells the compiler that the programmer wishes to reference the data types, procedures and functions defined in the package named after the *use* by the names given in the package definition without including the name of the package as a qualifier. In general, however, the user might not even have to enter these *with* and *use* clauses directly. The *use* and *with* clauses could be placed in the program template with which the user begins. Alternatively an *include pragma* could be added to the compiler which would read a file of *with* and *use* clauses and include them in the program. Both of these mechanisms would provide to the user an environment of data types and primitive operations tailored to a specific application, in this case robots.

```
with TEXT_IO;
with POSITION; use POSITION;                    -- Make the procedures, functions
with CAD_MODEL; use CAD_MODEL;                  -- and data types defined in the named
with VISION; use VISION;                        -- packages available to create a robot
with ROBOT; use ROBOT;                          -- environment for the programmer.
procedure MAIN is
    N: INTEGER;                                 -- Number in set of parts.
begin
    GET(N);                                     --input from terminal.
    declare
        •
        •
      SET_OF_PARTS: PART_SET (1..N);            -- Set of parts that could potentially
                                                -- appear on the conveyor.
      X: PART;                                  -- Data about the part found (4).
      TARGET_LOC, PICK: FRAME;                  -- Coordinate frames for the part
                                                -- and its grasp point (2).
        •
        •
    begin
      CALIBRATE;                                -- Calibrate the robot before starting (5).
        •
        •
      SET_SPEED (FAST);                         -- Set robot speed fast for motion
                                                -- to approach point (5).
      X:= FIND (DECISION_TREE (SET_OF_PARTS));
                                                -- Find and identify the part (3,4).
      PICK:= PICK_APP_POINT (X.NAME, X.STABLE_POS);
                                                -- Approach point from CAD d/base (3).
      TARGET_LOC:= X.LOCATION * PICK;           -- Express approach point
                                                -- in world coordinates (2).
      MOVE (TARGET_LOC);                        -- Move to approach point (5).
      SET_SPEED (SLOW);                         -- Set robot speed slow for final
                                                -- motion to grasp point (5).
      PICK:= PICK_POINT (X. NAME, X.STABLE_POS);
                                                -- Get grasp point from CAD database (3).
      TARGET_LOC:= X. LOCATION * PICK;          -- Put in world coordinates (2).
      MOVE (TARGET_LOC);                        -- Move to grasp point (5).
      CLOSE_GRIP;                               -- Grasp part (5).
    end;
end MAIN;
```

Figure 1. Outline of the Main Program Controlling the Robot.

The second half of the main program shows the use of the data types and functions provided by the Ada packages for the simple operation described above. The syntax used is similar to that found in several robot languages and the type and variable names are sufficiently mnemonic that one can follow the intent of the program with minimal reference to the supporting packages (see below).

Note that comments are introduced by a preceding "--" and they can be placed anywhere in the text stream. In addition, the comments in Figure 1 include one or two numbers in parentheses that are the figure numbers of relevant packages.

FIND is a procedure in the VISION package that finds and identifies the part on the input conveyor and returns the part's name, a 4×4 homogeneous transformation giving the location of a coordinate frame for the part in terms of the robot's world coordinates and an index of which stable position the part was found in. These three items of data are stored as components of a record X.

PICK_APP_POINT and PICK_POINT are functions which return (from the CAD database or utilities acting upon it) 4×4 homogeneous transformations which express the approach and grasp points in terms of the coordinate frame for the part. The "*" has been overloaded (see below) to mean multiplication of 4×4 matrices so that the result is the transformation of the appropriate point in terms of the world coordinates of the robot. TARGET_LOC holds this transformation and is the argument of the MOVE procedure which actually causes robot motion.

Partial specifications for the packages referenced in Figure 1 are given in Figures 2 through 5.

The POSITION package defines the type FRAME to be a 4×4 matrix for use as a homogeneous transformation (Figure 2). This type is intended to be used to represent various coordinate systems which will occur during the programming of a robot task in terms of other coordinate systems. While the 4x4 homogeneous matrix representation is most common for coordinate systems, it is not the only possibility. The POSITION package simply provides a standard interface to the programmers. The implementation can be changed, even placed in special hardware, without the robot programmer having to change any code. The use of the attribute *private* means that the programmer cannot use any knowledge of how the data types is to be implemented. The function definition "*" gives meaning to the operation * in the context of two variables of type FRAME. This process is called overloading of the operator *. The implementation of the function (not shown) will implement a multiplication of two 4×4 matrices. The special structure of the homogeneous transformation might be taken into account in the implementation, but this is of no concern to the programmer, who need only be concerned with using the function [Mud81, MVA82].

The package CAD_MODEL provides an interface to the off-line CAD system (Figure 3). This kind of package is not part of standard robot systems, but is an important part of our research on integrating robot programming and CAD. Several kinds of information can be derived from the CAD system. The vision system (see next section) calculates a set of features (area, perimeter, number of holes, etc.) from the image of the part and, to identify the part, uses a decision tree calculated from the set of parts which might be present. In other systems the decision tree is obtained by on-line training of the vision system. However, the decision tree can be precalculated from the part descriptions in the CAD database and stored for use by the programming system. Similarly, grasp points

```
package POSITION is

    type COORD is new FLOAT;
    type ANGLE is new FLOAT;
    -- COORD and ANGLE are declared "new" floating point types.
    -- This way they will not be confused with other FLOAT's.

    type FRAME is private;
    -- FRAME is the representation of one coordinate system in terms of another.

    function BUILD_FRAME(X,Y,Z: in COORD; R,S,T: in ANGLE) return FRAME;
    -- Allows FRAME's to be constructed from lower level primitives. Necessary
    -- since FRAME is private and its structure cannot be directly accessed.

    function  "*" ( A, B : in FRAME ) return FRAME;
    -- This function expresses the coordinate frame represented by B in
    -- terms of the one in which A is  represented, i.e., it is a transformation.

    procedure UNBUILD_FRAME(A: in FRAME; X,Y,Z: out COORD; R,S,T: out ANGLE );
    -- Complement of BUILD_FRAME.

private
    type FRAME is array (1..4,1..4) of FLOAT; -- A 4X4 homogeneous transformation.
end POSITION;
```

Figure 2. Package Specification for Coordinate Frames and Related Operations.

for the parts can be precalculated.

The functions of CAD_MODEL access the database holding the required values, and the data types defined provide the views of the data required by other packages. PART_ID and PART_SET provide data types for identifying one or a set of part(s). Each part will typically have a set of stable positions in which it may lie. These may also be determined off-line from the CAD database. The example shows the stable position identified by an integer index, though, since the type is private, this fact may not be used by the rest of the program. The stable position is part of the information returned by the VISION system and is used by PICK_APP_POINT and PICK_POINT to determine the relative position of the approach and grasp points of the part. The function STABLE_POS_SET returns the set of stable positions in which a given part may be found. DECISION_TREE returns the decision information which is used by VISION as the basis for distinguishing a set of parts from one another. The decision information is a binary tree pointed to by a (pointer) variable of *access* type DECISION.

The VISION package provides the interface to the vision subsystem (Figure 4). It uses the data types and interfaces provided by CAD_MODEL. The type

```
with POSITION; use POSITION;

package CAD_MODEL is
    type PART_ID is private;
    type D_INFO is private;
    type DECISION_INFO is access D_INFO;
    type PART_SET is array (INTEGER range <>) of PART_ID;
    type STABLE_POSITION  is private;
    type S_POS_SET is array (INTEGER range <> ) of STABLE_POSITION;
    function DECISION_TREE (S: in PART_SET) return DECISION_INFO;
    function STABLE_POS_SET (PART_NAME: in PART_ID) return S_POS_SET;
    function PICK_POINT (PART_NAME: in PART_ID;
                STABLE_POS: in STABLE_POSITION) return FRAME;
    function PICK_APP_POINT (PART_NAME: in PART_ID;
                STABLE_POS: in STABLE_POSITION) return FRAME;

private

        type PART_ID is new STRING (1..8);
                                        -- Eight character part identifier.
        type STABLE_POSITION is new INTEGER;
                                        -- Index of stable positions.

        type D_INFO is                  -- Node in binary tree.
            record
                VALUE: FLOAT;
                LLINK: DECISION_INFO;
                RLINK: DECISION_INFO;
            end record;
end CAD_MODEL;
```

Figure 3. Package Specification for CAD_MODEL.

PART that it defines has three components: the name of the part, a coordinate frame giving its location in terms of the world coordinates, and the stable position in which it was found. The function FIND causes a picture to be taken and returns a variable of type PART giving the pertinent information about the part found.

Finally, the ROBOT package provides a simple interface to the robot (Figure 5). The intended operation should be obvious from the procedure names chosen.

One principal advantage of this system is its modularity and extensibility. If a new sensing or algorithmic capability is added, one need only insert a new package for it, insert the appropriate *use* and *with* clauses to make the addition available to the user, and recompile the system. If one wishes to make the program available to run with a different robot (of sufficient physical capabilities to

```
with POSITION: use POSITION;

with CAD_MODEL; use CAD_MODEL;

package VISION is

    type PART is
        record
            NAME: PART_ID;
            LOCATION: FRAME;
            STABLE_POS: STABLE_POSITION;
        end record;

    function FIND (D_T: in DECISION_INFO) return PART;
    -- Identifies the part, its location and the position it is in.

end VISION;
```

Figure 4. Package Specification for VISION.

handle the problem) only the package ROBOT need be changed. A standardization of the package interface specification, then, could lead to ready availability of ROBOT packages for a wide variety of robots and simplify porting of programs from one robot to another.

IV. PROGRAM ABSTRACTION

A major objective of software management is in the ability to reuse algorithms that have been implemented previously. However, an undesirable attribute of strongly typed languages is the inability of type independent operations to be used on a variety of conflicting data types. Ada allows a form of program abstraction in which subroutines or packages (called generic subroutines or packages) take parameters which are the data types to be manipulated. The Ada source code is then expanded as necessary to implement the desired subroutine or package for each data type desired. The expansion process, called instantiation, is similar to macro expansion.

Generics are not limited, however, to operations which are independent of the data-types involved. Ada generics can adjust for broad classes of objects by using the *with* clause and by passing functions as parameters during instantiation. In this way operations which are specific to the data type given as a generic formal parameter can be inherited.

```
with POSITION; use POSITION;

package ROBOT is

   SLOW: constant := 0.1;                    -- Fine motion speed.
   FAST: constant := 1.0;                    -- Approach speed.
   subtype SPEED is FLOAT range SLOW..FAST;
                                             -- Bound speed for safety check.
   procedure CALIBRATE;                      -- Calibrate the robot arm prior to use.
   procedure MOVE(DESTINATION: in FRAME);
                                             -- Move to a point given by applying
                                             -- the transform represented by FRAME.
   procedure OPEN_GRIP;
   procedure CLOSE_GRIP;
   procedure SET_SPEED (SPD: in SPEED);

end ROBOT;
```

Figure 5. Package Specification for ROBOT.

The concept of generics raises several interesting questions. One is whether a generic robot package can be defined and instantiated for specific instances of different robots. To test this on a limited scale the use of generics in a vision system was examined. Contemporary vision systems utilize a list of nearly forty features which could be useful in distinguishing parts from one other, inspecting parts or guiding specific assembly operations. Most vision systems calculate subsets of four to twelve features from this list. Because of the limited amount of time, no vision system calculates the entire list at run-time. Selection of the proper subset is, therefore, crucial for an efficient solution. Furthermore, different applications require different subsets.

To obtain a vision system which could be easily adapted to a variety of vision tasks, a library of routines to calculate all features from an INTERMEDIATE_FEATURE_VECTOR could be established. Next, a generic FINAL_FEATURE_CALCULATION procedure could be written in which the FINAL_FEATURE_VECTOR is defined only in terms of the number of features desired. This is illustrated in detail in [VMG84]. However, little advantage was seen to be gained.

V. SUMMARY AND CONCLUSIONS

The future development of automated manufacturing cells will be increasingly linked to the integration of cell components amongst themselves and with

higher level computer aided engineering functions. This integration will depend upon increasingly complex and sophisticated computer systems. The programming of these systems will require modern software techniques such as data and program abstraction. This paper has outlined data and program abstraction use as the basis for developing manufacturing cell software and illustrated this with the implementation of a computer vision module via Ada.

From the view points of managing complex software, providing an application specific programming environment to the user, and achieving language standardization, Ada provides a number of advantages. These include:

- The use of data abstraction and operator overloading to create well modularized application specific code helps usability, readability and maintainability.
- The resulting application package can create a reasonable application specific environment.
- The strong type checking significantly aids debugging.
- The separate compilation features in conjunction with the other features above aids flexibility and helps portability.
- The expressive power of the language is excellent.

These advantages are not surprising. They are exactly what computer scientists have been predicting for several years. Having these capabilities widely available in a standardized language, however, is very significant. Indeed, it is this standardization of Ada that can greatly aid in standardizing application specific "languages" and giving them portability. The portability can be inherited, to a large measure, from Ada.

Generics, on the other hand, while of great use in dealing with common data structures over different primitive data types, was of less utility than originally expected in the application specific uses for which it was examined. It is possible to instantiate an application specific vision module, as shown above. Similarly, one could conceive of using generics to manage the production of code for different robots--just instantiate the code for the robot you want from some generic package. However, since in both the vision case and in the multiple robot case the controlling algorithms are different, one would have to pass in to the generic package (as parameters) the functions which perform the calculations specific to a given instantiation. While feasible, this eliminates much of the advantage to using generics. The resulting principle advantage would be an enforcement of a standard way of dealing with all features in the vision system or all robots in a multiple robot situation.

There are also a number of concerns which have arisen which either are a detraction to some users or bear further investigation (some of these reflect Ada implementation only, not data abstraction in concept):

- The heavy use of data abstractions creates additional procedure calls and corresponding overhead which can cause difficulty in a real-time environment.
- Strong typing can get in the way of what one wants to do.
- How usable will Ada really be, even with good environment creation through special packages, to the noncomputer professional?
- The debugging of robot programs requires close interaction with the programmer. It is not clear this can happen with Ada.
- The integration of systems involving multiple processors does not permit Ada communication and synchronization mechanisms to be fully utilized.

The use of the inline pragma was tested in our implementation of the vision system. By using *inline* for the most frequently used low level routines the computation time was reduced by a factor of nearly four. This must not be taken too seriously, however. The architecture of the iAPX 432, the computer used in our work, makes it particularly susceptible to inefficiency on context switching. Thus, the inline improvements in our experiments are probably much greater than will be obtained in general. Further investigation on the effectiveness of the expansion should be carried out. Also, the Ada inline pragma causes all invocations of a procedure to be expanded, while for memory management purposes, the programmer might find it more convenient to be able to selectively expand procedure calls.

The strong typing argument has raged for some time and is not specific to robot or manufacturing cell applications. We believe that as the size and complexity of a software project increase so does the importance of using strong typing.

We do not ever expect to see robots on manufacturing cells programmed in Ada by shop floor personnel. We expect that as more complex arrangements of robots, sensors and other machines are built and as better links with computer aided engineering and computer aided design database are forged, shop floor personnel will cease to "program" robots. Rather they will interact with a program to identify what is to be done next or which option to choose in responding to an exception. The actual programming will be done in a more generic fashion by a person who has a good mix of manufacturing and computer engineering/science in his/her background. A person with this type of training should be able to deal

with a "roboticized Ada".

The debugging issue is one that requires considerable additional research. All Ada implementations in progress are based on a compile translation while almost all robot programming languages are based on interpretive translation. From the point of view of the programmer, however, the robot program may be a separately prepared and debugged entity. What is really necessary is a fast interactive translate/debug system. This does not preclude compile translation, particularly if used in conjunction with a simulator.

Interprocess communication in our experimental system did not really use the synchronization mechanisms of Ada; the communication was necessarily handled through low level I/O drivers. This points to a major limitation in nearly all approaches to the integration of multiple smart devices, that is, the need to deal with all devices via explicit I/O and program the devices in (often) different languages (PL/M and assembly language in our case). Often the processes with which one wants to communicate or synchronize exist on separate processors and the language communication and synchronization mechanisms do not extend across machine boundaries. Consequently, we feel there is a strong need for a system integration language which can extend across machine boundaries. Whether or not Ada is suitable for such extensions is currently under investigation.

Recent programming language research has yielded a number of new concepts which will aid the program development process. Data and program abstraction have been incorporated into an experimental system through use of the Ada programming language. Experience to date with them has been quite favorable, though the need for a distributed language is beginning to emerge.

VI. REFERENCES

[Bar84] J. G. P. Barnes, *Programming in Ada*, (2nd Edition), London, England: Addison-Wesley, 1984.

[Bri77] P. Brinch Hansen, *The Architecture of Concurrent Programs*, Englewood Cliffs, NJ 07632: Prentice-Hall, 1977.

[DoD83] *Ada Programming Language (ANSI/MIL-STD-1815A),* Washington, D.C. 20301: Ada Joint Program Office, Department of Defense, OUSD(R&E), January 1983.

[Gle79] G. J. Gleason, "Vision Module Development," *Ninth Report, NSF Grants APR75-13074 and DAR78-27128, SRI Projects 4391 and 8487,* Stanford Research Institute, Menlo Park, CA, Aug. 1979, pp. 9-16.

[GoY80] D. I. Good and W. D. Young, "Generics and Verification in Ada," *Sigplan Notices*, Vol. 15, Nov. 1980, pp. 123-127.

[GSC82] W. A. Gruver, B. I. Soroka, J. J. Craig and T. L. Turner, "Evaluation of Commercially Available Robot Programming Languages," *Proceedings*

of the 13th International Symposium on Industrial Robots & Robots 7, April 1983, pp. 12-58 to 12-68.

[Hil80] J. W. Hill, "Survey of Commercial Vision Systems," *Industrial Automation Group*, May 1980.

[Mil78] R. Milner, "Theory of Type Polymorphism in Programming," *Journal of Computers and System Sciences*, Vol. 17, 1978, pp. 348-375.

[Mud81] T. N. Mudge, "Special Purpose VLSI Processors for Industrial Robotics," *Proceedings of the IEEE Computer Society's 5th International Computer Software & Application Conference*, Nov. 1981, pp. 270-271.

[MVA82] T. N. Mudge, R. A. Volz and D. E. Atkins, "Hardware/Software Transparency in Robotics Through Object Level Design," *Proceedings of the Society of Photo-optical Instrumentation Engineers Technical Symposium West*, SPIE 360, Aug. 1982, pp. 216-223.

[Org82] E. I. Organick, *A Programmer's View of the Intel 432 System*, Santa Clara, CA 95051: Intel Corp., 1982.

[Per81] W. A. Perkins, *A Computer Vision System that Learns to Inspect Parts*, General Motors Research Laboratories, Research Publication GMR-3650, June 1981.

[REM81] E. S. Roberts, A. Evans Jr., C. R. Morgan and E. M. Clarke, "Task Management in Ada - A Critical Evaluation for Real-Time Multiprocessors," *Software--Practice and Experience*, Vol. 11, 1981, pp. 1019-1051.

[Ren82] T. Rentsch, "Object Oriented Programming," *Sigplan Notices*, Vol. 17, No. 9, Sep. 1982, pp. 51-57.

[Sha80] M. Shaw, "The Impact of Abstraction Concerns on Modular Programming Languages," *Proceedings of the IEEE*, Vol. 68, No. 9, Sep. 1980, pp. 1119-1130.

[Shi82] K. G. Shin, *A Comparative Study of Robot Programming Languages*, Center for Robotics and Integrated Manufacture Report RSD-TR-17-82, Univ. of Michigan, Ann Arbor, MI 48109, Nov. 1982, 50 pp.

[VMG84] R. A. Volz, T. N. Mudge and D. A. Gal, "Using Ada as a Programming Language for Robot-based Manufacturing Cells," *IEEE Transactions on Systems, Man and Cybernetics*, (to appear).

[Wir82] N. Wirth, *Programming in Modula-2*, (Second Edition), Berlin, Germany: Springer-Verlag, 1982.

[WWV82] J. Wolter, T. C. Woo and R. A. Volz, "Gripping Position for 3D Objects," *Proceedings of the 1982 Meeting of the Industry Applications Society*, Oct. 1982, pp. 1309-1314.