

Using Ada as a Programming Language for Robot-Based Manufacturing Cells

RICHARD A. VOLZ, TREVOR N. MUDGE, AND DAVID A. GAL

Abstract—The Ada[®] programming language has been under development for the Department of Defense since 1976. Ada is intended as the Department of Defense's principal system implementation language. In particular, one of the primary aims of Ada has been to program real-time embedded systems. How Ada can be used to program a robot-based manufacturing cell, an example of a real-time embedded system is described in this paper. The computing issues in manufacturing cells are discussed with respect to Ada. Using an experimental manufacturing cell presently under construction as an example, a strategy for robot programming based on Ada is described. A case study of the software for the vision subsystem is used to illustrate a central feature of Ada: data abstraction. Additional important features of Ada for software management—program abstraction through generics and operator overloading, and multitasking—are also illustrated. The principal advantages and difficulties in using Ada for programming robot-based manufacturing cells are summarized based on the software issues described and the case study.

I. INTRODUCTION

WITH THE ADVENT of robot-based manufacturing cells, the need for a standard implementation language to program these cells has grown in importance. The present practice of designing new robot languages for nearly every new robot may satisfy the particular programming needs of each robot, but it is counterproductive from the standpoint of developing integrated manufacturing cell technology. Standardization is clearly needed. Moreover, the current high-level languages used to implement the real-time requirements of manufacturing systems lack some of the language tools, such as data abstraction, that facilitate programming in the large. Indeed, even the most sophisticated robot, numerically controlled (NC) machine tool, and related "manufacturing systems" programming languages presently in commercial use support neither data abstraction nor other features appropriate for large-scale programming [14], [28]. They are, therefore, unsuitable at the cell integration level. However, the Department of Defense's (DOD's) future system implementation language, Ada,[®] is an attempt to provide language constructs that can overcome most of these shortcomings.

Ada was originally developed at the instigation of the DOD [10] for programming embedded systems. Examples of embedded systems are, to quote one of the designers of

the language, "those for process control, missile guidance or even the sequencing of a dishwasher" ([2], p. vii). 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 extensions make Ada particularly appealing for programming large-scale real-time embedded systems—a situation characteristic of robot-based manufacturing cells. Though fully validated compilers have only recently become commercially available, DOD's strong support of the language guarantees a large-scale presence in the future. This therefore warrants a serious inquiry into the feasibility of using Ada as a standard implementation language for manufacturing cells.

This paper describes an initial effort at using Ada as the basis for programming part of a manufacturing cell, specifically a robot, an interface to a vision sensor, and an interface to a computer-aided design (CAD) system. The system exploits the features of data abstraction, separate compilation, multitasking, exception handling, encapsulation, operator overloading, and generics found in the language; it expands on work summarized in [30]. Some of the principal computing issues involved in programming such a system are identified and discussed vis-à-vis the use of Ada. Much of the discussion is based on the example of the vision subsystem developed and the limitations and improvements encountered in the process. Section II discusses computing issues in manufacturing cells and how they relate to Ada. Section III describes appropriate features of Ada and a strategy for using Ada. Section IV illustrates one of the central features of Ada, data abstraction, by examining the development of the vision subsystem software. Section V discusses how program abstraction in the form of generics and operator overloading can be used to manage program development. Section VI discusses how multitasking can be used. Section VII summarizes the experience to date and notes both important advantages of using Ada and areas of concern where more work is needed.

II. COMPUTING ISSUES IN MANUFACTURING CELLS

To illustrate some of the computing problems that arise in flexible manufacturing cells, consider the simple cell shown in Fig. 1, major portions of which are presently

Manuscript received September 15, 1983; revised March 27, 1984. This work was supported in part by the Zimmer Foundation and in part by the Air Force Office of Scientific Research under contract F49620-82-C-0089.

The authors are with the Robot Systems Division, Center for Robotics and Integrated Manufacturing, Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109.

[®] A registered trademark of the Department of Defense.

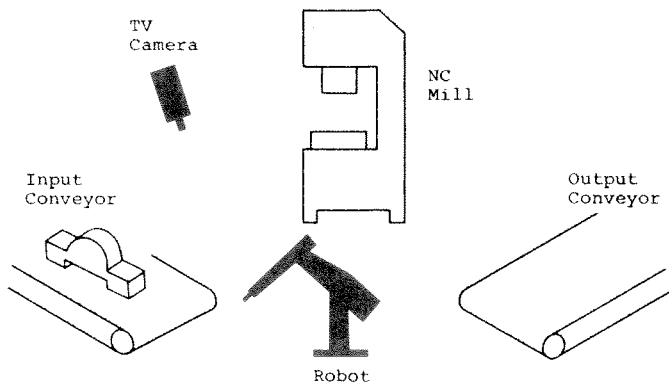


Fig. 1. Manufacturing cell.

under development in the Robot Research Laboratory at the University of Michigan. This cell is a simple machine loading/unloading system consisting of a process machine (an NC milling machine for this example); an input conveyor; a camera and associated machine vision system to sense incoming parts; a robot for loading, unloading, and tool changes; and an output conveyor. The incoming stock may arrive randomly oriented and must be located and identified by a machine vision system using the camera before being grasped by the robot. This example is conceptually relatively straightforward and, in fact, omits some additional features one might realistically expect such as post-machining inspection and adaptive control of the machining process. It is sufficient, however, for illustrating the principal points to be discussed.

The above example can also be used to illustrate a related activity that we believe will be of great importance in the future and that 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 the machine vision system [15], [12], [24]. 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 via a CAD system.

The computer system that manages this cell must interact with a variety of devices and with at least one level of external computer system. The complexity of the resultant system, the interaction with real-time devices, and the interaction with the external computer system create the principal computing problem. A hierarchical computer system for controlling the cell is shown in Fig. 2¹. A central cell control computer manages the overall behavior of the cell and handles communications with the CAD database

(in the cell being developed this is a multiprocessor version of Intel's iAPX432). Dedicated microcomputers function as attached processors to the cell controller and interface with the various physical process machines and sensors in the cell (in the cell being developed these are typically Intel 8086 based microcomputers). Even if all the necessary algorithms were known, which is far from the case [17], there are significant computing issues remaining:

- the development and management of the complex real-time software system to control the cell;
- the extraction of information from the CAD system to assist cell operations;
- the architecture of the computer system (hardware and software) to support the cell;
- the computational speed to meet the real-time requirements.

This paper primarily addresses the first of these points. The second point is beyond the scope of this paper although it does impinge upon the case study that we present. Research into the use of CAD information for cell operations, beyond automatic production of NC tool programs, is still in its infancy. Some early work in this area can be found in [4], [5], [33]. The last two issues are also beyond the scope of this paper; more on them can be found in [19], [20], [21], [29].

The software aspect of robot cell control itself encompasses a number of issues. Among the more important are

- the management of large complex software systems;
- the efficiency of code produced for real-time applications;
- interprocess communication and task synchronization;
- portability;
- program debugging, particularly the real-time aspects.

The first of these issues becomes more important when one considers complex systems with sophisticated sensors and several robots rather than a single dumb robot. Such systems will be much more common in the future, and the need to manage the ensuing software complexity will be correspondingly greater. The proven way of dealing with complexity in any system, software or otherwise, is to partition it into subsystems whose complexity is manageable. This partitioning is important not only in the design phase but also for system maintenance, where, in the case of embedded systems, it has been shown that often over 70 percent of the lifetime cost of large embedded software goes into software maintenance [7].

The second of these issues, computation speed, is a function not only of the underlying hardware but also of the efficiency of the code emitted by the translator. Several software techniques have evolved for dealing with this issue, two of the most important being code optimization and in-line code substitution. The first will almost certainly be developed for any translator used widely by the military or industry. The second can be particularly important when data abstraction causes a large number of procedure

¹Similar hierarchical structures have been proposed by several authors, e.g., Albus [1] and Wisnosky [34].

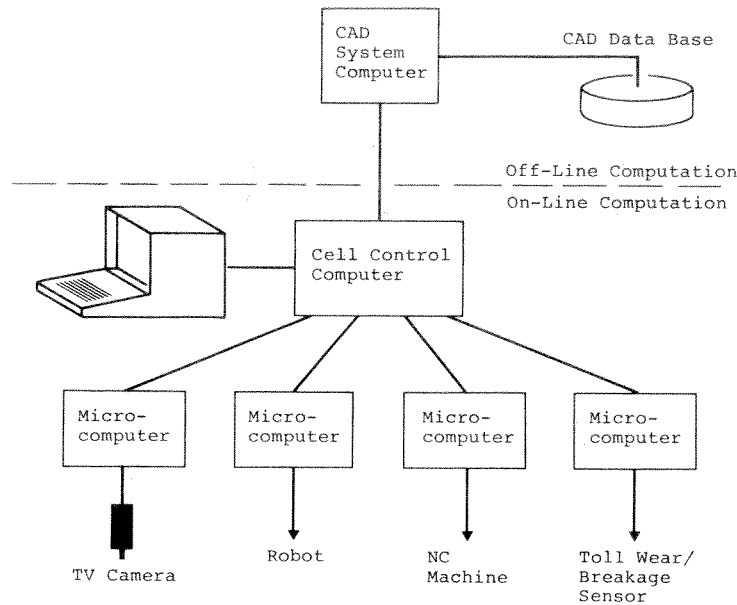


Fig. 2. Cell control.

calls (as it does in Ada). Nevertheless, the effectiveness of these, particularly for a language as large as Ada, is a matter of serious concern.

The third of these issues, interprocess communication and task synchronization, arises out of a need to support a multiprocessing system (see Fig. 2). Communication and synchronization are required in various multiprocessing situations: among tasks executing on the same processor, between a central control process and a dedicated attached processor controlling some sensor or robot joint, and among tasks executing on separate general purpose processors. Ideally, a robot programming language should support communication and synchronization mechanisms.

The fourth of these issues, portability among different robots and computer systems, is one of the principal difficulties with most robot programming languages. Little has been done to address this problem. It is most important that future language efforts not be made heavily machine dependent.

Finally, the fifth issue, program debugging, is complicated by the real-time aspect [31]. Reconstructing real-time events to track down errors is usually difficult and often impossible because of the irreproducibility of the external real-time events. Further, due to the fact that the physical movement of the devices under computer control can be very slow (at least compared to computer speeds), painfully long debug times can result. Unfortunately, the debugging issue cannot be ignored because the occurrence of an error that cannot be halted in midstream can be both dangerous and expensive.

The features present in Ada permit many of these issues to be addressed. The following section sketches a general way in which Ada can be tailored for expressing robot, sensor, and CAD operations.

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 multi-tasking 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 design goals of Ada directly address most of the issues raised above, and these will be discussed in the context of their use for the programming system. The strategy and example described are based on the experimental system being implemented at the University of Michigan. The system will be used as an experimental vehicle for testing algorithms linking CAD to robot and sensor operation 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

structure² having a unique identifier and an associated set of functions and procedures that can operate on it [22]. 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. In effect, the data type can be abstracted and known only through operations performed on it. Thus, object-based programming provides a way to implement data abstraction, which, as a result of work in programming language design during the 1970's, has emerged as a major organizational concept in programming languages [27]. 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 [32] and Concurrent Pascal [6], Ada is the first that is likely to see widespread use.

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 [13], [18], 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 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 [27]. In addition to forming the basis for real-time operations, tasks also provide a means of increasing processing efficiency in a parallel 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.

²The meaning of the term "object" is not universally agreed upon; our usage is fairly narrow. See [26] for a discussion of various viewpoints.

The program and data abstraction capabilities of Ada can give rise to a large number of procedure calls that in turn add processing overhead to the program. Ada provides an *inline pragma* (a compiler directive that 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 or robot programming based upon Ada depends upon three central ideas:

- 1) the use of Ada's extensibility;
- 2) the use of Ada's data and program abstraction;
- 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 primitives 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 system consisting of a robot, a vision sensor and a link to a CAD data base (e.g., part of the system shown in Figs. 1 and 2) is considered in a simplified manner. 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 data base 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.

Fig. 3 shows a portion of a sample main program. The action part of the example shown is to find a part on the 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 reprograms 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 (see Fig. 2). 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

```

with TEXT_IO;
with POSITION; use POSITION;
with CAD_MODEL; use CAD_MODEL;
with VISION; use VISION;
with ROBOT; use ROBOT;

procedure MAIN is

    N: INTEGER;

begin

    GET(N);
    declare
        .
        .

        SET_OF_PARTS: PART_SET (1..N);

        X: PART;
        TARGET_LOC, PICK: FRAME;

        .
        .

    begin

        CALIBRATE;
        .
        .

        SET_SPEED (FAST);

        X:= FIND (DECISION_TREE (SET_OF_PARTS));
        PICK:= PICK_APP_POINT (X.NAME, X.STABLE_POS);
        TARGET_LOC:= X.LOCATION * PICK;

        MOVE (TARGET_LOC);
        SET_SPEED (SLOW);

        PICK:= PICK_POINT (X. NAME, X.STABLE_POS);

        TARGET_IOC:= X. LOCATION * PICK;
        MOVE (TARGET_LOC);
        CLOSE_GRIP;
    end;

end MAIN;

```

-- Make the procedures, functions
-- and data types defined in the named
-- packages available to create a robot
-- environment for the programmer.

-- Number in set of parts.

--input from terminal.

-- Set of parts that could potentially
-- appear on the conveyor.
-- Data about the part found (6).
-- Coordinate frames for the part
-- and its grasp point (4).

-- Calibrate the robot before starting (7).

-- Set robot speed fast for motion
-- to approach point (7).
-- Find and identify the part (5,6).
-- Approach point from CAD d/base (5).
-- Express approach point
-- in world coordinates (4).
-- Move to approach point (7).
-- Set robot speed slow for final
-- motion to grasp point (7).

-- Get grasp point from CAD database (5).
-- Put in world coordinates (4).
-- Move to grasp point (7).
-- Grasp part (7).

Fig. 3. Outline of the main program controlling the robot.

are omitted since they would not 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). Such utilities are, in fact, under development and nearly complete [33].

The first part of this example identifies Ada packages that 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 a program template with which the user begins. Alternatively an *include pragma* could be added to the compiler that would read a file of *with* and *use* clauses and include them in the program. In this way an environment of data types, and primitive operations tailored to a specific application, in this case robots, could be provided to the user.

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 Fig. 3 include one or two numbers in parentheses that are the figure numbers of relevant packages.

```

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;

```

Fig. 4. Package specification for coordinate frames and related operations.

```

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;

```

Fig. 5. Package specifications for CAD_MODEL.

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 that return (from the CAD database or utilities acting upon it) 4×4 homogeneous transformations that 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 that actually causes robot motion.

Partial specifications for the packages referenced in Fig. 3 are given in Figs. 4–7.

The POSITION package defines the type FRAME to be a 4×4 matrix for use as a homogeneous transformation (Fig. 4). This type is intended to be used to represent various coordinate systems that will occur during the programming of a robot task in terms of other coordinate systems. While the 4×4 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 implementa-

```

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;

```

Fig. 6. Package specification for VISION.

```

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;

```

Fig. 7. Package specification for ROBOT.

tion 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.

The package CAD_MODEL provides an interface to the off-line CAD system (Fig. 5). 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 being identified and uses a decision tree calculated from the set of parts which might be present to identify the part. Normally, the decision tree is obtained by on-line training of the vision system. However, the decision tree can be precalculated from the part description in the CAD data base and stored for use by the programming system. Similarly, grasp points for the parts can be precalculated [33].

The functions of CAD_MODEL access the database holding the required values, and the data types defined pro-

vide 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 that is used by VISION as the basis for distinguishing elements of a set of parts from one another. The decision information is a binary tree pointed to by a variable of access type DECISION.

The VISION package provides the interface to the vision subsystem (Fig. 6). It uses the data types and interfaces provided by CAD_MODEL. The type 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 object found.

Finally, the ROBOT package provides a simple interface to the robot (Fig. 7). 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 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 a much easier porting of programs from one robot to another.

IV. DATA ABSTRACTION

In the previous section the use of the separate compilation and the package facilities of Ada to create a robot programming environment were illustrated. In this and the next section the use of Ada is viewed from a different perspective: that of incorporating modern software concepts such as data and program abstraction into the robot programming environment. If properly used these features can bring several additional advantages to the robot programming system, including

- clearer conceptualization of the problem being programmed;
- better data security and avoidance of side effects;
- easier modification of the implementation;
- better maintainability and readability of the code.

These advantages, of course, are not intrinsic to Ada, but may be achieved to varying degrees in different languages by good programmers. However, Ada does provide the mechanism to make their use convenient. As robot programs become larger and encompass more of the manufacturing cell, these advantages will rise sharply in importance.

To provide a more detailed study of the use of Ada data and programming abstraction for cell control, a case study of one subsystem of the cell, the CAD-based vision system, is explored.

A. Overview of the Vision System

The vision subsystem was developed to allow recognition of nonoverlapping parts in our experimental manufacturing cell. It was not intended to produce new algorithms for computer vision, nor was it intended to be a simple transliteration of an existing vision system into Ada code. Rather, the goals of the vision subsystem were to implement the Stanford Research Institute (SRI) vision algorithms, taking advantage of the facilities provided in the Ada programming language to explore both the use of Ada and the use of CAD information to replace the vision training phase.

The SRI vision algorithms are described in [12] (for additional important concepts see [9], [11]). They are in-







Area	
Perimeter & P^2/Area	
Number of Holes Hole Area	
Orientation Angle & Centroid	
Bounding Box Length & Width	
Maximum, Minimum & Average Distance from Centroid	

Fig. 8. Typical silhouette features.

tended to classify nonoverlapping parts represented as dark silhouettes against a light background. The image used is a matrix of binary valued pixels. The vision algorithms compute a set of features from the silhouettes. These include area, A ; perimeter, P , and P^2/A ; number of holes and hole area; position of centroid and its angle of orientation with respect to the field of view; bounding box length and width; maximum, minimum and average distance of the perimeter from centroid (see Fig. 8). A subset of the features (determined by the specific vision task at hand) are used to classify the parts viewed. The type of visual recognition that the SRI system is capable of is clearly very restricted. However, the system represents a judicious mix of well-understood techniques that are versatile enough for many industrial applications and that can be easily implemented in a small, rugged low-cost package. Indeed, systems based upon these techniques are now being produced by several vendors.

The connection of the vision system to the skeleton robot programming environment actually requires more detail than illustrated in Section III. The *with* clause must be expanded to include other packages that are referenced by the function of VISION. Fig. 6 is to be modified to

```
with CAD_MODEL; use CAD_MODEL;
with ANALYZE; use ANALYZE;
package VISION is
    ...
end VISION.
```

The package ANALYZE provides the procedures and functions that calculate the features of the part to be recognized.

Actually, there are several levels of hierarchical decomposition within ANALYZE. It references three packages that are collectively responsible for determining the features of the object. Its specification begins

```

package ANALYZE is
  procedure SCANLINE_ENCODER (...);
  procedure CONNECTIVITY_ANALYSIS (...);
  procedure FINAL_FEATURE_CALCULATIONS (...);
  :
end ANALYZE.
    
```

The implementation (not shown) of FIND in VISION uses SCANLINE_ENCODER, CONNECTIVITY_ANALYSIS and FINAL_FEATURE_CALCULATIONS to determine the part features needed to use the decision tree. The following discussion will show how Ada has been employed to achieve the benefits of modern software concepts.

B. Ada-Based Implementation of Feature Extraction

1) *Operation Overview:* ANALYZE operates on a binary matrix of pixels that forms the image. The image is processed in raster scan order (one row at a time). Each raster scanline is converted from a row of light and dark

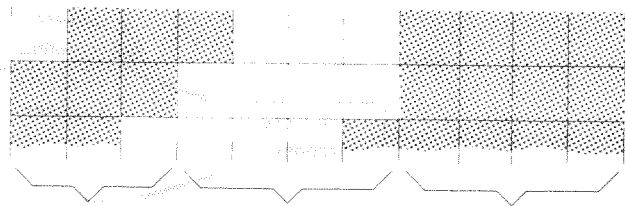


Fig. 9. Scanlines of pixels.

maintained. Thus, when all the runs for a part (hole) have been linked together the intermediate feature vector and perimeter list contain the information that is needed to calculate the final feature values for each part.

2) *Object-Based Modularization:* Program modularization using Ada objects is based upon creating an access module for each data structure that must be used by all procedures or functions referencing the data objects. Fig. 10 shows the relation of the data entities, the access modules and the routines provided by ANALYZE. The double headed hollow arrows indicate module access to data structures. These modules are the only ones that need to know the implementation details of the data structure. The data structures (see the right column of Fig. 10) operated on by the feature extraction program are as follows:

IMAGE	—
ENCODED_SCANLINE	—
INTERMEDIATE_FEATURE_VECTOR	—
PERIMETER	—
FINAL_FEATURE_VECTOR	—

- The camera's 256 × 256 pixel image.
- Expanded run length encoding of a scanline which includes the values needed to calculate the first and second moments, etc.
- Values accumulated from the ENCODED_SCANLINE's.
- Contains all the exterior pixels for a given part (hole).
- Scalar values for each feature of each part and hole in the scene.

pixels to a smaller sequence of "runs" of contiguous pixels of a particular color. For example, the center scanline depicted in Fig. 9 has three runs of 3 (dark), 4 (light), and 4 (dark). Thus, the scanline can be "run length encoded" by the sequence 3-4-4 with one additional bit to indicate that the first run is dark. Dark runs are interpreted as belonging to a nonoverlapping part, while light runs are considered empty space surrounding the part or holes in the part.

Next, consecutive scanlines are analyzed for connectivity, overlapping dark runs and overlapping light runs are linked to form parts and holes respectively. The consecutive scanlines of Fig. 9 are linked into three regions: two potential parts separated by a gap or hole. As part of the linking or region growing operation, the number of pixels in each region (area), and the first and second moments of each region are calculated. This is accomplished by adding the values for each scanline to partial sums being accumulated in an intermediate feature (hole) vector. In addition, a list of the perimeter pixels of each part (hole) is

The operation of the feature calculation can be viewed, as follows: SCANLINE_ENCODER (SLE) obtains lines one at a time from the IMAGE module and encodes them, and, through the SCANLINE module stores the encoded scanlines. The scanlines are next accessed by CONNECTIVITY_ANALYSIS (CNA) that performs analysis of connected regions. CNA processing of one scanline can be done concurrently with the generation of the encodings of the scanline by SLE. The operation of FINAL_FEATURE_CALCULATION (FFC) can be similarly overlapped with that of CNA. The clear conceptualization afforded by this view of the operation exposes the fact that logically the above three subprograms can be considered to be operating in a pipelined fashion. In Section VI it will be shown how this logical pipelining could be translated into actual pipelining using Ada tasking.

By way of contrast, Fig. 11 shows a counterpart of Fig. 10 that would be typical of the more common control-based approach in which the program is based upon the flow of

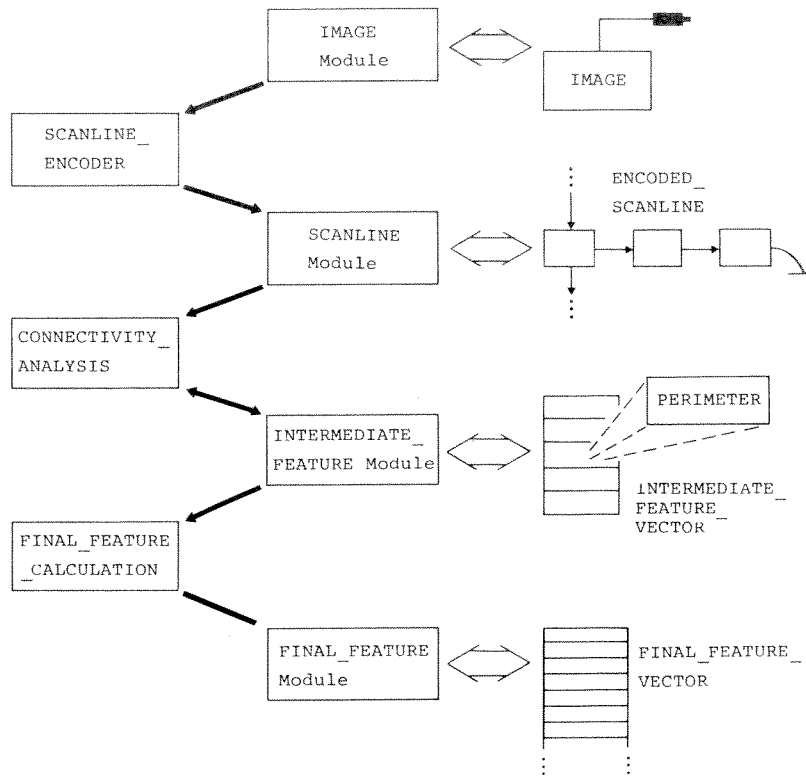


Fig. 10. Object-based modularization.

control. The more convoluted diagram obscures the conceptual simplicity apparent in the object-based approach. The procedures and functions in this case directly access the data structures they need. Data security is thus violated and possibility of spreading the effects of programming errors is increased (unwanted side effects).

3) Modification and Maintainability of the Program:

The maintainability of the program is also substantially enhanced by the use of object-based modularization in Ada. The separation of the specification and implementation parts of a package provides a localization in procedure implementation changes. Both data structure and code changes are simplified. When modifications are required to the data structure, the only code that must be modified is in the local modules that have exclusive access to the data. For example, it is unimportant for the program to know the actual implementation of the IMAGE module, as long as there is a means of finding the coordinates where the image changes from light to dark and dark to light. The specification for the IMAGE package appears as

package IMAGE is

```

LOAD_NEW_IMAGE (...)  --- Command camera to
                        take a picture and
                        return results.
NEXT_LIGHT_PIXEL (...) --- Return coordinates.
NEXT_DARK_PIXEL (...)  --- Return coordinates.
IMAGE end;
```

modules used have similar simplicity and mnemonic association with the required data access functions.

The above capability was used to considerable advantage during the debugging of the system. In actual operation, LOAD_NEW_IMAGE causes a GE-TN2500 camera to load an image into a frame grabber; an attached processor then translates the resulting matrix into a run length encoding and transfers the encoded image to the iAPX432. The vision system was ready for debugging, however, before the frame grabber was completed. To debug the system a simple package substitution was made: the IMAGE module access routines were changed to request column and row information from an operator's terminal. This proved to be an effective way to debug the system.

C. Objects in Ada

Object-based programs are rather difficult to implement in traditional block structured high-level languages. For example, data structures that must be accessed by separate

which clearly has the needed access functions. The other blocks must be visible to at least one block within which

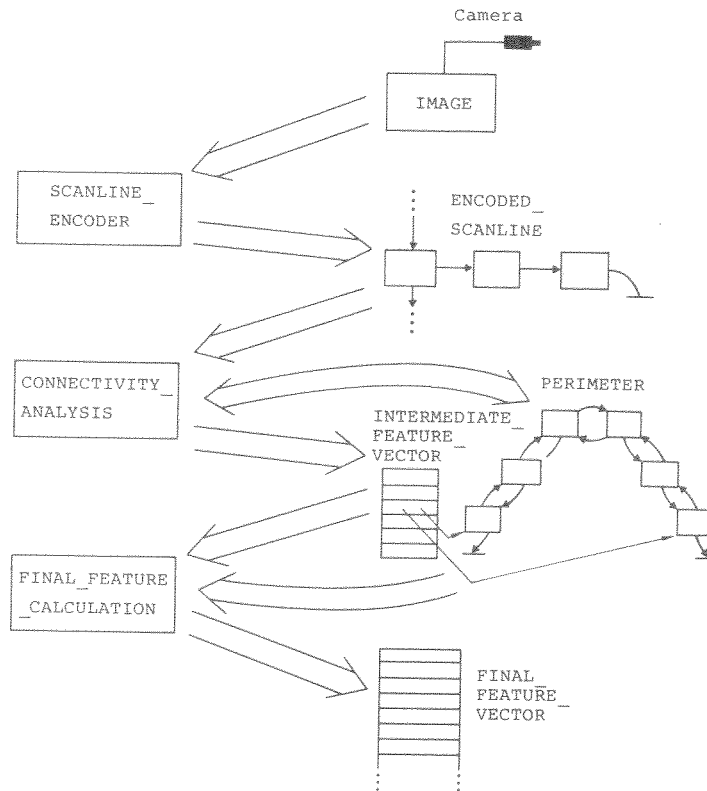


Fig. 11. Typical control-based modularization.

the separate blocks are nested. This leaves open the possibility of other nested blocks having unintended access to the data structures. In less restrictive languages memory locations can be made available to the subprograms that need access to them; however, this again raises the question of undesirable accessibility. In contrast, objects in Ada can be encapsulated with the package construct. Controlled access to the object is defined by the specification part of the package that formally defines visibility and scope rules for variable data objects and subprograms, and creates a visible environment for both the package body and any modules outside the package that need to manipulate objects in the package. Therefore, the specification is the construct that facilitates abstraction and permits easy implementation of abstract data types. The package body completes the code for each procedure and function in the package specification. The body and any modules that reference the package can be compiled separately from the specification, as long as they are able to recreate the environment of the specification section during their own compilation. In this way strong static-semantic checking can be achieved concurrently with separate compilation.

V. 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 one to write a form of subroutine or package,

(called generic subroutines or packages) in which the data-types manipulated are formal parameters. 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 macroexpansion.

Generics are not limited, however, to operations which are independent of the data types involved. Though not as flexible as polymorphic functions [18], Ada generics can adjust for broad classes of objects by use of the *with* clause [10] 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. For example, if the `FRAME` type of the package `POSITION` were used as an argument to a generic procedure, the "*" would inherit the meaning of matrix multiply.

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 the vision system was examined. Contemporary vision systems [3], [15] 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.

```

generic
  type FEATURES is (<>); --enumeration of desired features
package FINAL_FEATURES is
  type FEATURE_VECTOR is array (FEATURES) of float;
end FINAL_FEATURES;

--Next we define the generic feature calculation procedure

generic

  type FEATURES is (<>); --enumeration of desired features
  type FINAL_FEATURE_VECTOR is private;
  type INTERM_VECTOR is private;
  with function FEAT_CALC_1 (INTERM: in INTERM_VECTOR) return float;
  .
  .
  with function FEAT_CALC_N (INTERM: in INTERM_VECTOR) return float;

procedure FINAL_FEAT_CALC(IMFV: in INTERM_VECTOR; FFCV: out FINAL_FEATURE_VECTOR);
procedure FINAL_FEAT_CALC(IMFV: in INTERM_VECTOR; FFCV: out FINAL_FEATURE_VECTOR) is

  INDEX: FEATURES; --index into FINAL_FEATURE_VECTOR

begin
  INDEX := FEATURES'FIRST;
  FFCV (INDEX) := FEAT_CALC_1(IMFV);
  .
  .
  INDEX := FEATURES'SUCC(INDEX);
  FFCV(INDEX) := FEAT_CALC_N(IMFV);
end FINAL_FEAT_CALC;

--Next, we show a sample instantiation of final feature calculation

type DESIRED_FEATURES is (FEAT1, .. , FEATN);

package FINAL is new FINAL_FEATURES(DESIRED_FEATURES); --This creates a package
--with the desired feature vector.

procedure FINAL_FEATURE_CALCULATION is new FINAL_FEAT_CALC(
  DESIRED_FEATURES, --This now exists from the above subtype.
  FINAL_FEATURE_VECTOR, --This exists from the above package instantiation.
  INTERMEDIATE_FEATURE_VECTOR, --This is defined elsewhere in the vision system.
  FEATURE_CALC_1, --This must be previously defined.
  .
  .
  FEATURE_CALC_N); --This must be previously defined.

--Actual performance of the feature calculations occurs upon the following call assuming
--X and Y suitably declared earlier

FINAL_FEATURE_CALCULATION(X,Y);

```

Fig. 12.

To obtain a vision system that 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. (See Fig. 10 to relate module names to ensuing discussion.) 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 Fig. 12. The generic procedure is called FINAL_FEAT_CALC. Its operation is expressed primarily in terms of a set of individual feature calculation functions generically called FEAT_CALC_1, ..., FEAT_CALC_N. To create an actual instance of FINAL_FEAT_CALC that can be used to calculate a final feature vector, one must identify the set of features desired (accomplished by type DESIRED_FEATURES in the example) and supply actual objects for the generic parameters of FINAL_FEAT_CALC. FEATURE_CALC_1, ..., FEATURE_CALC_N in the example, must be separately prepared. The instantiation creates an actual procedure FINAL_FEATURE

_CALCULATION that can be used to calculate the final feature vector. The point, of course, is that one can create different final feature calculations procedures by supplying different feature calculation functions.

While it was originally envisioned that through the use of generics a variety of relatively specific and efficient vision systems could be produced at a minimal programming cost, it is not clear that a great deal could be gained. One still must write all of the individual routines for feature calculation. Generics simply provides one way of incorporating the specific set to be used. Moreover, the example of Fig. 12 requires advance knowledge of the number of features to be used and is not amenable to use of a loop construct in implementation. These two restrictions might be eliminated by implementing the feature calculation routines as tasks and making one of the generic parameters to FINAL_FEAT_CALC a record whose components contain access pointers to the individual feature tasks. The record could have a discriminant stating the

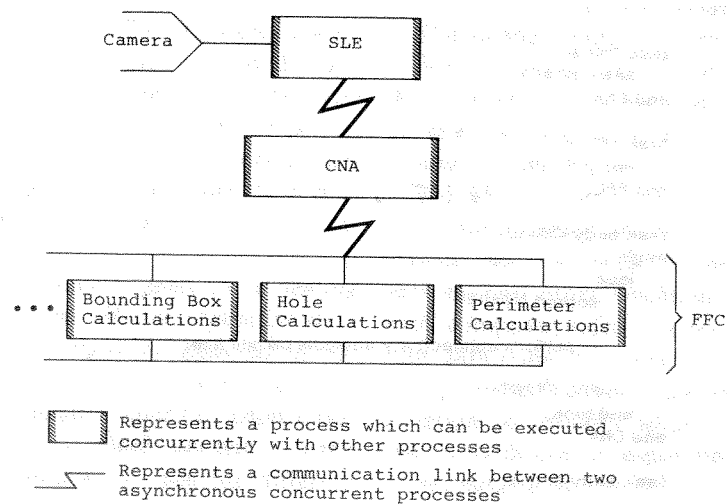


Fig. 13. Multitasking.

number of components. However, it hardly seems worth the extra complexity. The use of generics, then, to build a specially tailored vision system does not seem to offer as much as originally anticipated.

VI. MULTITASKING

The cell control computer (the iAPX432) in our experimental cell has a multiprocessor architecture that automatically provides parallel computation for multiple concurrent processes. Coupled with Ada's explicit multiprocessing through the task construct, it is possible to easily speed up some computation through parallel processing. This is illustrated for the vision system.

Again, using the ANALYZE package as an example, consider the processing of the three subprograms for scanline encoding (SLE), connectivity analysis (CNA), and final feature calculation (FFC). The clear conceptualization of Fig. 10 suggests that the calculation can be pipelined, and through tasking and multiprocessing, can be converted to parallel computations. Instead of waiting for all of the scanlines to be entirely encoded, CNA can begin after SLE has processed a single scanline. While CNA is processing one line of data, SLE can generate the next encoded scanline. A similar process can take place between CNA and FFC. Thus, conceptually each of the three tasks can be considered to be executing in a loop: fetching information from the previous task and sending information to the following task (see Fig. 13).

This pipelining can show a substantial reduction in processing time. With pipelining the processing time has a lower time bound equal to the time to process a single data item by all the tasks (filling the pipeline up) plus the time for the slowest task to work on $k - 1$ data items where k is the total number of data items processed. Let $T(k)$ be the time to run a program on k data records without multitasking, and let $MT(k)$ be the time to run the same program (now composed of n tasks, P_1, \dots, P_n) with

multitasking, then

$$MT(k) = \text{time}(P_1 + P_2 + \dots + P_n)$$

$$+ (k - 1) * \text{time}(\max(P_1, P_2, \dots, P_n)) \leq T(k)$$

assuming sufficient processors are available. This, of course, is an ideal situation that assumes uniformity in the processing of data items and little or no overhead to manage the extra processors. Nevertheless, programs that cycle through large data sets on systems with enough processors should gravitate to this lower bound.

Transmission overhead is incurred with every transaction from the attached processor. This transmission overhead can also be overlapped with the operation of the SLE task, effectively hiding the overhead.

The calculations in the third task, the final feature calculations (FFC), conveniently separate into independent subtasks such as bounding box calculations, hole calculations, perimeter calculations, etc., all requiring the same input, the INTERMEDIATE_FEATURE_VECTOR, and all outputting to unique locations in the FINAL_FEATURE_VECTOR. These independent calculations can be grouped into parallel subtasks within the FFC task. The parent task (FFC) merely passes chunks of work to each subtask and waits for them to finish (see Fig. 12). Processing time for the tasks is bounded by the time necessary to process a data item through the longest subtask. In other words

$$MT(k) = \text{time}(\max(P_1, P_2, P_3, \dots, P_n)) \leq T(k)$$

where P_1, \dots, P_n are the n tasks that can run in parallel. Again this is an ideal situation, but processing using this technique should reflect significant speed up provided the necessary processors are available.

The structure of an example Ada procedure corresponding to Fig. 13 that illustrates the pipeline and parallel computations possible is shown in Fig. 14.

```

procedure SLE is

  task CNA is
    entry START;
  end CNA;

  task FFC is
    entry START;
  end FFC;

  task body CNA is
  begin
    loop
      accept START;

      -- perform connectivity analysis on one line;

      FFC.START;
    end loop;
  end CNA;

  task body FFC is
  begin
    loop
      accept START;

      declare

        task FC1;

        task FCN;

        task body FC1 is
        begin
          --perform feature calculation update on one line of data
        end FC1;

        task body FCN is
        begin
          --perform feature calculation update on one line of data
        end FCN;
        begin
          --just let the feature tasks run in parallel
        end;
        end loop;
    end FFC;

  begin -- SLE

    loop -- until done
      -- perform one line of scan line encoding analysis
      CNA.START;
    end loop;
  end SLE;

```

Fig. 14.

VII. SUMMARY AND CONCLUSION

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 DOD language, Ada, was developed specifically for large complex real-time embedded software systems. This paper has outlined its use as the basis for developing manufacturing cell software and illustrated this with the implementation of a computer vision module via Ada. Five issues in robot cell software were identified at the beginning of the paper: complexity, code efficiency, communication and synchronization, portability, and debugging. Our experience to date has touched upon a majority, but not all, of these issues. The following paragraphs summarize our experience and conclusions about them.

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 following.

- 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 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 that 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:

- 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 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 databases 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 [8], [16], [23].

Interprocess communication has been investigated in two contexts, the multitasking version of the vision system described in the previous section and attached processors for low level vision and robot control. The effectiveness of the former depends upon multiple processors with the capability to automatically pick up tasks and execute them as they are created. The latter 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, 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 that 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. A number of these are incorporated into Ada. Future languages will undoubtedly encompass more of these concepts. However, at present, the considerable resources being put into the Ada effort by the DOD coupled with its orientation toward real-time embedded systems makes us believe it will be a significant factor in

the future. While we are not yet prepared to state that Ada is the answer to robot and manufacturing cell programming, we have been pleased with it so far and feel further investigation is warranted.

REFERENCES

- [1] J. S. Albus, Charles R. McLean, Anthony J. Barbera, and M. L. Fitzgerald, "Hierarchical control for robots in an automated factory," in *Proc. 13th Int. Symp. Industrial Robots and Robots*, vol. 7, Apr. 1983, pp. 13-43.
- [2] J. G. P. Barnes, *Programming in Ada*. London, England: Addison-Wesley, 1982.
- [3] H. G. Barrow and J.M. Tenenbaum, "Computation vision," *Artificial Intelligence Center, Stanford Research Institute*, vol. 69, May 1981, no. 5, pp. 572-95.
- [4] E. W. Baumann, "Model based vision and the MCL language," in *Proc. IEEE Syst. Man, Cybern. Conf.*, Oct. 1981, pp. 433-438.
- [5] E. W. Baumann, "CAD model input for robotic sensory systems," in *Proc. Autofact IV*, Nov. 1982.
- [6] P. Brinch Hansen, *The Architecture of Concurrent Programs*. Englewood Cliffs, NJ: Prentice-Hall, 1977.
- [7] J. D. Cooper, June 17-18, 1982, "Why a DoD standard programming language," in *Proc. Ada Conf.*, Boston, MA and Washington, D.C., June 28-29, 1982, pp. 1-6.
- [8] Ruediger Dillmann, "A graphical emulation system for robot design and program testing," in *Proc. 13th Int. Symp. Industrial Robots and Robot 7*, Apr. 1983, pp. 7-1-7-15.
- [9] R. Duda, J. Kremers, and D. Nitzan, "Automatic part classification," Stanford Research Institute, Menlo Park, CA, Fifth Rep., SRI Project 4391, Jan. 1976, pp. 7-22.
- [10] *Ada Programming Language (ANSI/MIL-STD-1815A)*, Washington, DC 20301: Ada Joint Program Office, Department of Defense, OUSD(R & E), Jan. 1983.
- [11] R. Duda and D. Nitzan, "Error analysis for automatic part recognition," 5th Rep., NSF Grant G138100X1, SRI Project 4391, Stanford Research Institute, Menlo Park, CA, Jan. 1976, pp. 65-81.
- [12] G. J. Gleason, "Vision module development," 9th Rep., NSF Grants APR75-13074 and DAR78-27128, SRI Projects 4391 and 8487, Stanford Research Institute, Menlo Park, CA, Aug. 1979, pp. 9-16.
- [13] D. I. Good and W. D. Young, "Generics and verification in Ada," *Sigplan Notices*, vol. 15, Nov. 1980, pp. 123-127.
- [14] W. A. Gruver, B. I. Soroka, J. J. Craig, and T. L. Turner, "Evaluation of commercially available robot programming languages," *Proc. 13th Int. Symp. Industrial Robots & Robots 7*, Apr. 1983, pp. 12-58-12-68.
- [15] J. W. Hill, "Survey of commercial vision systems," *Industrial Automation Group*, May 1980.
- [16] S. J. Kretch, "Robotic animation," *Robots VI Conf.*, Mar. 1982, (not in Proceedings: see McDonnell Douglas Automation Co., St. Louis, MO.).
- [17] T. Lozano-Perez, "Robot programming," Artificial Intelligence Lab, MIT, Cambridge, MA, Dec. 1982, A.I. memo no. 698, pp. 56.
- [18] R. Milner, "Theory of type polymorphism in programming," *Comput. and Syst. Sci.*, vol. 17, pp. 348-375, 1978.
- [19] T. N. Mudge, "Special purpose VLSI processors for industrial robotics," in *Proc. IEEE Computer Society's 5th Int. Computer Software & Application Conf.*, Nov. 1981, pp. 270-271.
- [20] T. N. Mudge, R. A. Volz, and D. E. Atkins, "Hardware/software transparency in robotics through object level design," in *Proc. Soc. Photo-optical Instrumentation Engineers Technical Symp. West*, SPIE 360, Aug. 1982, pp. 216-223.
- [21] E. I. Organick, M. P. Maloney, D. Klass, and G. Lindstrom, "Transparent interface between software and hardware versions of Ada compilation units," Univ. Utah Rep. UTEC-83-030, Dept. Computer Science, Univ. Utah, June, 1983, 19 pp.
- [22] E. I. Organick, "A Programmer's View of the Intel 432 System", Intel Corp., Santa Clara, CA, 1982.
- [23] Alan de Pennington, M. Susan Bloor, and Mazin Balila "Geometric modelling: A contribution towards intelligent robots," in *Proc. 13th Int. Symp. Industrial Robots and Robots 7*, Apr. 1983, pp. 7-35-7-54.
- [24] W. A. Perkins, A Computer Vision System that Learns to Inspect Parts, General Motors Research Laboratories, research publication GMR-3650, June 1981.
- [25] 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, pp. 1019-1051, 1981.
- [26] T. Rentsch, "Object oriented programming," *Sigplan Notices*, vol. 17, no. 9, pp. 51-57, Sept. 1982.
- [27] M. Shaw, "The impact of abstraction concerns on modular programming languages," *Proc. IEEE*, vol. 68, no. 9, pp. 1119-1130, Sept. 1980.
- [28] K. G. Shin, A Comparative Study of Robot Programming Languages, Center for Robotics and Integrated Manufacturing, Univ. of Michigan, Ann Arbor, MI rep. RSD-TR-17-82, Nov. 1982, pp. 50.
- [29] J. L. Turney and T. N. Mudge, "VLSI implementation of a numerical processor for robotics," in *Proc. 27th Int. Instrumentation Symp.*, Indianapolis, IN, Apr. 1981, pp. 169-175; also presented at the Instrument Society of America Anaheim Conf., Anaheim, MI, Oct. 1981.
- [30] R. A. Volz, T. N. Mudge, and D. A. Gal, "Using Ada as a robot system programming language," in *Proc. 13th Int. Symp. Industrial Robots & Robots 7*, Apr. 1983, pp. 12-42-12-57.
- [31] R. A. Volz and R. E. Richardson, *CRASH User's Manual*, Dept. Electrical and Computer Engineering Rep., Univ. of Michigan, Ann Arbor, MI, Aug. 1977.
- [32] N. Wirth, *Programming in Modula-2*, 2nd ed. Berlin, Germany: Springer-Verlag, 1982.
- [33] J. Wolter, T. C. Woo, and R. A. Volz, "Gripping position for 3D objects," in *Proc. 1982 Meeting of the Industry Applications Soc.*, Oct. 1982, pp. 1309-1314.
- [34] Dennis E. Wisnosky, "Computer integrated manufacturing: The Air Force ICAM approach," SME/CASA tech. paper M581-953, 1981.