USING ADA AS A ROBOT SYSTEM PROGRAMMING LANGUAGE

by

R.A.Volz, T.N.Mudge and D.A.Gal
Robotics Research Laboratory
Department of Electrical and Computer Engineering
University of Michigan
Ann Arbor, MI48109
313/764-4343

ABSTRACT

The ADA programming language was developed for the Department of Defense in 1978. ADA is intended as the DoD's sole system implementation language, in particular, one of the primary aims of ADA is for the programming of real-time embedded systems. This paper will describe the logical structure of an ADA based multiprocessing system being used for the real-time control of a robot system. The robot system is comprised of a PUMA 600 arm and a vision subsystem that uses a GE TN2500 camera as an input device. It will be shown how multiprocessing tasks in the robot system can be easily achieved using ADA running on a dual processor machine. Using multiprocessing to meet real-time constraints will also be discussed. The paper will emphasize the research associated with the vision subsystem.

Preprint from the conference proceedings of the 13th International Symposium on Industrial Robots and Robot 7, April 17-21 1983, Chicago, pp. 12-42 thru 12-57.

I. INTRODUCTION[1]

With the advent of flexible automated manufacturing environments, the need for a standard implementation language to program manufacturing cells has become important. The present practice of designing new robot languages for nearly every new robot is certainly counterproductive in the long run [1]. Moreover, current high level languages seem insufficient for the real-time requirements of production systems, while standards for the sake of readability, flexibility and portability are absent from most manufacturing languages. Design requirements for the Department of Defense's (DoD's) future system implementation language, Ada [2], assumes all of these properties. Though fully validated compilers are not currently commercially available, DoD's strong support of the language guarantees a large scale presence in the future. Therefore, a serious inquiry into the feasibility of using Ada as a standard implementation language for manufacturing is warranted.

This paper describes the logical structure of an Ada-based manufacturing cell, exploiting the data-abstraction and multitasking features of the language. Specifically, the discussion will focus on the vision subsystem developed in Ada and the limitations and improvements encountered in the process.

II. SOFTWARE ISSUES IN MANUFACTURING CELLS

To illustrate some of the problems which arise in flexible manufacturing cells consider the simple cell illustrated in Figure 1 (figures are after the bibliography). This cell is a simple machine loading/unloading system consisting of an NC milling machine, an input conveyor, a robot for loading, unloading and tool changes, and an output conveyor. For purposes of generality we assume that the incoming stock arrives in a disoriented fashion and must be located and identified by a machine vision system before being grasped. The goal is for the cell to be fully automatic and flexible in the sense that it can be used to manufacture any part within some reasonable class. There is to be no loss of production time due to the training of either the robots or the machine vision system as occurs presently [3], [4]. It is presumed that all the parts to be produced have been designed via a CAD system.

A hierarchical computer system controls the cell as shown in Figure 2. Dedicated microcomputers interface with the various physical process machines and sensors in the cell. These communicate with a central cell control computer which manages the overall behavior of the cell and handle communications with the CAD database. The principle computer issues are:

- extraction of information from the CAD system to assist cell operations
- development and management of the complex real-time software system to operate the cell
- the architecture of the computer hardware/software system to support the cell
- computational speed

This paper addresses principally the second of these problems. The last two are beyond the scope of this paper and are treated elsewhere [5]-[7]. Exploration of the use of CAD information for cell operations beyond automatic production of NC programs is only beginning, and is also treated elsewhere [8]-[10], though portions of our work on automatic training of vision systems impinge upon the current discussion.

As the cell complexity grows, so too does the software system which must manage it. In similar cases it has been shown that over 70% of the total cost of developing embedded software goes

[1] This work was supported in part by the Zimmer Foundation and in part by the Air Force Office and Scientific Research under contract F49620-82-C-0089.

into software maintenance [11]. It thus becomes critical to treat the software development functions very carefully. Ada was designed to address such problems through its support of abstract data types, separate compilation, multi-tasking, and extensibility.

The use of Ada for cell control is explored through a case study of a portion of the cell: the operation of the CAD driven vision subsystem. The underlying philosophy of Ada is centered upon the use of objects for program design. **Objects** are essentially data structures[2] [12]. The key element is that they maybe embedded in **packages** and made visible to program parts that would use them only through procedure and function references. That is, object and packages together hide data structure implementation and are an implementation of abstract data types. However, since objects are only visible through the operations performed on them, we will use the term object oriented loosely to imply the object/package combination. **Generic** procedures, an attempt at polymorphic function implementation [14], [15] allow operations to be defined over a set of data types, thus providing a broader use of objects. **Tasks** provide a means of dividing a program into logically concurrent operations with possible synchronization between them [16]. In addition to forming the basis for real-time operations, in a parallel processor environment, they may also provide a means of increasing processing efficiency. While there are many other features in Ada, these are the principal ones used in the vision system.

III. DESIGN OF THE VISION SYSTEM

The goal of the vision module project was to write a vision subsystem in Ada to provide a vision interface for our experimental manufacturing cell and allow recognition of nonoverlapping parts. VISUAL (Vision Interface System Using the Ada Language) was not intended to produce new algorithms nor was it intended to be a simple translation of existing vision systems into comparable Ada code. Rather, the goals of VISUAL were to implement the SRI (Stanford Research Institute) vision algorithms [4], taking advantage of the facilities provided in the Ada programming language, both to explore the use of Ada and the use of CAD information to replace the vision training phase.

The main implementation issue behind the development of VISUAL was to create a vision subsystem which could be used extensively for the variety of future projects concerning research with manufacturing cells. It was not only important to make VISUAL transportable to a variety of systems and configurations, but also extensible for a variety of vision tasks to be performed. Therefore, clear modularization, documentation and data abstraction were built into the system to allow future users to easily tailor VISUAL to their particular problems.

The overall vision module can be divided into three main submodules and a vision driver which communicates with an external program or an operator at a terminal (see Figure 3). The following are brief descriptions of the modules to familiarize the reader with the techniques used in solving the vision problem, and to set the stage for discussions on the use of Ada.

Feature Extraction

The feature extraction module is based on binary input from a video camera [4]. Raster scan lines are converted from an array of white and black pixels to a smaller sequence of bands which represent a run-length encoded sets of pixels which are of the same color and in contiguous locations on the line. If silhouettes are used, dark bands are interpreted as projections of a part while light bands are considered to be the absence of a part (i.e., a hole or empty space).

---

[2] The definition of "objects" is problematic, and our usage is fairly narrow. See [13] for a discussion of various viewpoints.

VISUAL analyzes consecutive scanlines, linking overlapping dark bands to form blobs and overlapping gap bands to form holes. Concurrently, values corresponding to the number of pixels and horizontal and vertical coordinate summations are calculated. Then when the links are made, these values are added into appropriate locations in an intermediate feature vector. In addition, concurrent computation of perimeter pixels are calculated. Thus when all the bands for a blob have been linked together, the intermediate feature vector and perimeter structure already contain the information which is needed to calculate the final feature values. These features, which usually contain values for area, perimeter length, number of holes, etc., are then used to identify the part in the scene.

Decision Tree Builder

Prior to the actual operation of the vision system, it is necessary to train the system to recognize the parts expected. Because of the quantization of the image, distortion through perspective and deviations within the limits of manufacturing tolerances, a given part will usually have mean and variance values for each feature measured [17], [18]. These values can be used to construct a decision strategy to follow at run-time.

To generate this mean and variance information, the vision system need only to be given a series of views of a part from a variety of locations in the viewing area. Traditionally, manual training of the vision system is required before each run-time situation [3], [19]. The work being done at the University of Michigan (as is also being done elsewhere [10]) generates the images from CAD information, calculating the necessary statistics off-line.

For each class of parts to be expected, a set of subclasses can be formed on the basis of the single feature which possesses the widest variance between mean values of the feature over the class of parts (measured in units of standard deviations). Thresholds can be calculated which divide the given set into subsets which should be disjoint and cannot be equivalent. Repeating this process on each subclass of parts produced until no classes can be further divided (i.e., each consists of only one part) yields a decision strategy for the run-time operation.

This decision strategy is then implemented through a decision tree data-structure. Each subtree's root node consists of the identity of the feature to be tested and then the thresholds which direct the routine to the corresponding offspring nodes. Each leaf node contains the part identified by the respective path through the structure. This essentially formulates a decision at run-time with roughly log(n) scalar comparisons.

Decision Mechanism

This module processes a given feature vector which has been calculated from an input image through the decision tree created prior to run-time as described above. It then sets up a resulting data structure which can return such information as the identity of the part, coordinates of its centroid and angle of orientation as well as possible pick points for robot acquisition.

IV. Object-Oriented Construction of VISUAL

One of the most crucial problems facing the designers of embedded computer systems is the management of complex software systems. As manufacturing becomes more and more integrated with sophisticated technologies, the level of software complexity dramatically increases. The need for such extensive software will require techniques to improve software organization, reliability and maintainability as programs are modified and expanded to fit changing needs. It has long been

established that the solution for these problems is in efficient program modularization.

Conventional modularization

Good programmers initially factor problems into general functions to be performed by the solution program. These operations usually constitute logical processing units within the program and may later become subroutines or functions which are called in specified sequences by a driver routine. Typically these processes act on subsets of the global data structures, generating desired output from a stream of input. The global data-structures correspond to COMMON blocks in a FORTRAN program.

The feature extraction module of the vision system, may be broken down in the following way:

SLE ::= Scan-Line Encoding
CNA ::= Connectivity Analysis
FFC ::= Final Feature Calculation

Thus a single run through the program would be the processing of one image by each module in succession.

One way of defining the set of data structures operated on by the program is the following:

Frame ::=
Camera's 256X256 pixel array.
Band_Line ::=
Expanded run-length encoding represented as a linked-list of blob and gap bands.
Intermediate Feature Vectors ::=
Arrays containing summation values of area, coordinate summations, perimeter values, etc., for each hole and blob in the scene.
Perimeter ::=
Doubly linked-list which contains all the exterior pixels for a given part.
Final_Feature_Vector ::=
Vector which contains the scaler values for each feature requested.

Each process of the program takes subsets of these data-structures as parameters for their operation. A data-structure used as input for a process will be prefaced by the word **in** and a data-structure output from a process will be prefaced by the word **out.** Thus the above program becomes:

program(**in** Frame, **out** Final_Feature_Vector) ::=

SLE(**in** Frame, **out** Band_Line vector)
CNA(**in** Band_Line, **out** Intermediate_Vector list, **inout** Perimeter)
FFC(**in** Intermediate_Feature_Vector, **in** Perimeter, **out** Final_Feature_Vector)

As we can see in Figure 4, both the CNA and FFC processes must have direct access to both Intermediate_Feature_Vector and Perimeter structures. Therefore, design decisions as to how Intermediate_Feature_Vector and Perimeter were implemented must be known by both. In the case where this design needs to be changed, which often occurs in the lifetime of a well used piece of software, we are faced with the task of patching each process in the program which uses the structures to correct the situation. Obviously, in very large systems with extensive

interdependence of data-structures, this could be a tremendous task.

Object modularization

The problems encountered with conventional methods of modularization had little to do with the algorithms for calculating the information. They lay with the storage and retrieval of the information once it was generated. In addition, debugging a conventional program is made more difficult when a single data structure is accessed and modified by the entire program; memory can be *clobbered* from anywhere and tracing offending statements is a formidable proposition. To avoid these problems, we must hide these design decisions from the processes which only need to know values and not necessarily how these values are stored and retrieved.

Program modularization based upon Ada objects allows us to create a module which has sole access to a given data structure. All external processes must access data through rigidly defined storage and retrieval subroutine calls. Then when modifications are required in the data structure, the only code which must be modified are the local routines which access the data. For example it is unimportant for the program to know the actual implementation of the Frame object, as long as there is a means of finding the coordinates of where the image changes from white to black and black to white. Similarly, the program does not need to know how the band lines or final objects are implemented, only how to store and retrieve information.

In addition we would like to have processes interact with objects at a uniform level of abstraction. Instead of processes knowing that an intermediate feature vector and its perimeter, as in Figure 4, are distinct, we treat the perimeter as another attribute of the intermediate feature object by embedding a perimeter module with in an Intermediate Feature Module, as shown in Figure 5. Thus the feature extraction data-structures can be factored into the following object modules.

Frame Module:
  dump_new_frame()--command camera to take a picture and return results
  next_black_pixel() return coordinates
  next_white_pixel() return coordinates

Band_Line Module:
  store_blob(<band-info>)--storage details not needed by user
  store_gap_band(<band-info>)
  next_band(band) return band--starting coordinate to length
  retrieve_<attr>(band) return <attr>--<attr>=hole or blob

Intermediate Feature Module:
  store_features(<feature-info>)
  sum_<attr>(feature_vector,value)
  retrieve_<attr>(feature_vector)return<attr>
  start_perimeter(<pixel_coordinates>) returns perim_pixel
  link_perimeter(feature_vector,perim_pixel) returns length
  retrieve_<attr>(perim_pixel) return <attr>

Final Feature Module:
  store_<feature>(<feature-info>)--e.g. area, perimeter, centroid
  retrieve_<feature>(final_vector) return <feature>

The interface is now clearly defined and hence easier to use by the program, and, most importantly more easily maintained and modified (compare Figures 4 and 5).

Objects in Ada

Object oriented programs are a rather difficult to implement in traditional high-level languages. Data structures which must be accessed in separate blocks must be global to the block containing both in most highly structured languages. In less restrictive languages, the memory locations can be made available to the subroutines which need access to them, but then we are faced with problems of information security. In Ada, modules for both objects and processes are clearly defined with the package constructs. A package, as the name implies, contains a collection of routines and related data structures, composing a syntactic and semantic unit [2]. A package definition consists of a specification section and a package body. The specification formally defines visibility and scope rules for variables and data objects as well as the subprograms which may be called. It creates a global environment for both the package body and any routines which need to access the module. The body and any modules which reference the package can then be compiled separately from the specification as long as they are able to recreate the environment of the specification section during their own compilation. The package body elaborates the code for each subroutine and function in the package specification. In this way, strong static-semantic checking can be achieved concurrently with separate compilation. Packages are then used to create data and process modularization.

As illustrated above, we abstracted the data-structures into data-objects to hide their actual implementation. Each object contains a set of routines which perform storage operations and access information contained in the objects. By making the data structures exclusive to the object module, using the Ada construct **private**, user processes are forced to use only these routines when accessing the object. The inefficiencies produced by excessive subroutine calls can be reduced with the INLINE pragma (a compiler directive which expands the subroutine source code inline wherever called).

The most immediate feature of such large scale data modularization aside from facilitating program development is tremendous program flexibility. By compiling a package body separately from its specification, we can fix the interface of processes with a module without elaborating any code for the actual implementation. Thus, when entire programs are linked, different package bodies can be substituted to provide a variety of interfaces. To illustrate the power of this flexibility we will elaborate the work accomplished with the frame module.

The frame buffer contains a single load routine which conceptually causes a new image to be dumped into the frame object and two access operations which return the row and column of the next occurrence of a black or white pixel, essentially returning a binary run-length encoded output of an image. In the run time operation of VISUAL, the first operation causes a GE-TN2500 camera to dump an image into a frame grabber, and then an attached processor translates the resulting matrix into a run length encoding. The encoded image is then dumped into the execution machine and the access instructions return the encoded stream as called.

However, because the actual implementation is hidden from the rest of the system, we can replace the package body for the frame buffer and have entirely new processing modes. In debugging the system, it is necessary to enter images interactively. Through a simple package substitution, the frame buffer access routines request column and row information from an operator's terminal. This allows us to bypass the need to perform elaborate calculations on entire frames of information initially and test specific calculation problems directly. In addition, experiments with replacing manual training using real images of an object by images generated from CAD information can be easily performed. By substituting another body for the frame buffer, we could read images created from CAD information on a host mainframe.

Tailoring VISUAL with Generics

An undersirable attribute of strongly typed languages is the inability of type independent operations to be used on a variety of conflicting data-types. An example which is often mentioned is a storage management module. Objects of the same data-type are to be dynamically pushed and popped in the manner of a queue or stack. The manager does not need to know any information about the object being stored, it only places a new object in the first free location and retrieves the front-most or top-most object. The generic facilities of Ada allow one to define such a module and then expand the Ada source code necessary to implement it for each data type desired. One merely writes generic subroutines or packages, leaving the data-types manipulated by the package as formal parameters. One can then create an instance of the package which is specific to a given data-type in the compilation of a package. This allows repetitive software to be written once and then copies which are specific to a given data-type can be created for each instance.

Generics are not limited, however, to operations which are totally independent of the data-types involved. Though not as flexible as polymorphic functions [14], Ada generics can adjust for broad classes of objects due to the **with** clause [2]. Because certain operations which are specific to the data-type given as a generic formal parameter can be inherited, we can extend the use of generics to more elaborate software systems such as robot vision.

In the process of developing the vision module, it was unclear as to what features should be calculated to best suit our problems. Contemporary vision systems [3], [20] utilize a list of nearly forty features which could be useful in distinguishing between objects, 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 effective solution. Furthermore, different operations require different subsets. General part recognition usually focuses on total area, perimeter and first or second moments of inertia while vision systems tailored to operations such as valve inspection [21] are obviously more concerned with hole information.

We wanted a vision module which could be easily adapted to the variety of expected vision tasks in our manufacturing cell. It was important to build-in a mechanism which would facilitate modifying the feature set to be calculated while at the same time allowing for fast run-time execution. We, therefore, explored the possibility of using Ada generics to instantiate (create an instance of) vision systems tailored to a specific subset of features, thereby placing most of the code generation burden on the compiler.

A library of routines to calculate all features from given object vectors was established. To create an instance of the vision system using a subset of the features, one creates a feature vector module which contains the definition of only the feature vector to be used and a feature calculation driver which makes calls to the necessary feature calculation routines. This is done through generic formed parameters. In this way, relatively specific and efficient vision systems can be produced at a minimal programming cost.

When this technique is generalized to the entire manufacturing cell, a great deal of extensibility is added to cell programs. Not only are the programs transportable, pending advent of Ada compilers for many systems, but also reconfigurable to the various machines if their interfaces are implemented in an object-oriented manner. Thus a manufacturing cell program which performs a given cell task with a generic robot arm, a comparable program can be developed for a system with an Asea or Puma robot as long as the interface has been designed according to a generic specification, simply by generating an instance of the program specific to the robot type. This, of course, requires a greater amount of effort at development time but may payoff if the program will be

widely used.

Speed Up Through Multitasking

With ever diminishing costs for hardware, multiprocessor systems will become more and more cost-efficient. Not only does parallel processing expand the complexity of problems which can be solved in real-time, but also allows for greater fault tolerance in operation. The manufacturing cell's central computer already must synchronize activities in attached processors controlling motion in robot arms, performing low level vision operations from camera input and executing tooling programs in an NC device. In addition, the cell's computer could also have a multiprocessor architecture, (e.g. the Intel 432) providing more powerful computer control and computation. Finally, processes which absorb exorbitant processing time may need to be down loaded to special purpose hardware made possible through VLSI technology. An implementation must make such a decision transparent to the user program.

Unfortunately, conventional programming languages lack formal constructs to facilitate programming in a multiprocessor environment. Defining separate processes and then establishing communication and synchronization is a nontrivial programming operation. Plus, most solutions require so much knowledge of the hardware targeted for execution that transportability is virtually impossible.

Conversely, the Ada **task** construct provides a relatively clear means of dividing a program into processes which can be executed in parallel on one processor or concurrently on several different processors [16]. Tasks are declared in a similar manner as subroutines. Communication is formally defined with message passing facilities. Whether or not this leads to any real ease in parallel programming depends upon both the computer implementation and the underlying computer architecture. In our experimental cell an Intel 432 system is used which automatically handles parallel processing of tasks.

Again using the feature extraction module as an example, suppose the actual operation was to continually read images from the camera and execute the program on each. If we define each process as a task instead of as a subroutine, we can take advantage of the extra processing power available in our multi-processing system. The scan-line encoding process can read images from the camera and then send Band_Lines to the connectivity process. Now, instead of waiting for the rest of the scan lines to be entirely encoded, the connectivity analysis begins while the scan-line encoding process fetches the second raster scan line. Thus each process performs its own loop, fetching information from the previous process and sending information to following processes (see Figure 6). This technique, identified as "serial concurrency," shows a reduction in processing time which has a lower time-bound of the time to process a single record once plus the amount of time for the slowest process to work on k-1 records where k is the total number of records processed.

Let T(k) be the time to run a program on k data records, then:

$$Tsc(k)=time(\ P1+P2+...+Pn\ ) + (k-1)*time(max(P1,P2,...,Pn)) <= T(k)$$

where n is the number of processes in the program, assuming sufficient processors are available. This, of course, is an ideal situation which assumes uniformity in the processing of records and little or no overhead to manage the extra-processors, but programs which cycle through large data sets on systems with enough processors, should gravitate to this lower bound.

This particular technique, aside from maximizing the use of processors also provides a degree of flexibility in the run-time implementation. The scan-line encoding task must be performed on every raster line in the image. The transmission and calculations on this array prove to be to be quite exorbitant. Therefore, instead of having the image dumped directly into the central cell computer, the major portion of the work is done in an attached processor which works directly with the low level frame grabber. Because the program had been divided into tasks originally, the final

implementation required no special interface with the rest of the processes.

The calculations in the third process, the final feature calculations, easily separate into independent subtasks such as bounding_box_calculations, hole_calculations, perimeter_calculations, etc., all requiring the same input, the Object_Buffer, and all outputting to unique locations in the final feature vector. These independent calculations can be grouped into parallel subtasks within the process. The parent process merely passes chunks of work to each subtask and waits for them to finish (see Figure 6). This method, identified as "parallel concurrency," shows a reduction in processing time with a lower bound of the time necessary to process each record through the longest subtask.

$$Tpc(k) = time(max(Sp1, Sp2, Sp3,...,Spn) <= T(k)$$

where Spi is the ith parallel sub-process out of n processes. Again, this is in an ideal situation, but extensive processing using this technique should reflect significant speed-up. Not only do we reduce the processing but with enough processors we could add additional features to our chosen subset without increasing processing time.

Further, by defining the library of subtasks to be used with the feature calculation module, we have established an efficient mechanism for implementing the generic tailoring of the vision system. For a given subset of features, we write a feature calculation process which spawns the subprocesses necessary to calculate the given feature set. Then when the vision system is instantiated via the appropriate feature vector as a formal parameter, this process, specific to the feature vector, is inherited via the Ada **with** clause.

V. SUMMARY AND CONCLUSIONS

The future development of automated manufacturing cells will be increasing linked to the integration of cell components amongst themselves and with higher level computer aided engineering functions. This integration will depend upon increasing complex and sophisticated computer systems. Ada was developed specifically for large complex real-time embedded software systems. This paper has explored its use as the basis for developing manufacturing cell software and illustrated this with the implementation of a computer vision module via Ada.

The use of Ada packages to implement abstract data types (objects) was shown to provide a clean, easily read implementation. The well defined interfaces to the data and hiding of implementation details make it easy to shift from one implementation to another, either for debugging purposes, or to accommodate new developments in associated hardware. The use of generics helps one to write general systems and then take tailor them to specific configurations for given problems. When Ada is used in conjunction with an object based multiprocessing hardware system, the use of tasking in unconventional ways can yield a parallel programming implementation of some algorithms, thus increasing the overall through put. Though not explicitly discussed here, the ability to define one's own data types and overload (that is redefine) operators on datatypes can be a significant aid in developing readable programs.

One of the principal goals of the use of Ada as the base of programming for the manufacturing cell is to achieve highly readable and easily maintained code. While it is too soon to assess the maintainability of the code, it can be noted that the use of Ada substantially enchanced the code production process. The programmer was totally unfamiliar with Ada, SRI vision work, and the hardware system upon which the code was to run at the beginning of that period of time. Moreover, a significant portion of time was lost due to bugs in both the hardware and software of the system. In spite of these difficulties the program which constituted 5,000-6,000 lines of code was produced in less than four months.

Another issue, one that requires considerable additional research, is that of debugging. Ada is intended for compile based translation and almost all robot programming languages are interpretive. From the point of view of the cell programmer, however, the robot program may be a separately prepared and debugged entity. Moreover, what is really necessary is a fast interactive translate/debug system, and this does not preclude compile translation, particularly if used in conjunction with a simulator [22].

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 undoubtably encompass more of these concepts. However, at present, the considerable resources being put into the Ada effort by DOD coupled with its orientation toward real-time embedded systems make it a candidate for manufacturing cells worth investigating. Based on the work outlined here it is our conclusion that from a programming point of view the use of Ada will be a highly attractive alternative.

VI. BIBLIOGRAPHY

[1] K. G. Shin, "Comparative Study of Robot Programming Languages," Center for Research in Integrated Manufacturing Report, (to appear).

[2] *Reference Manual for the Ada Programming Language (Revision)* Washington, D. C.: United States Department of Defense, July 1982.

[3] J. W. Hill, "Survey of commercial vision systems," *Industrial Automation Group*, May 1980.

[4] 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, pp. 9-16, Aug. 1979.

[5] J. L. Turney and T. N. Mudge, "VLSI implementation of a numerical processor for robotics," in *Proc. of the 27th Int. Instrumentation Symp.*, Indianapolis, IN, pp. 169-175, April 1981, also presented at the Instrument Soc. of America Anaheim Conf., Oct. 1981.

[6] T.N. Mudge, "Special Purpose VLSI Processors for Industrial Robots," in *Proc. of the IEEE Computer Soc. 5-th Int. Computer Software & Application Conf.*, pp. 270-271, Nov. 1981.

[7] T. N. Mudge, R. A. Volz, and D. E. Atkins, "Hardware/Software transparency in robotics through object level design," in *Proc. of the Soc. of Photo-optical Instrumentation Engineers Technical Symp. West*, Aug. 1982.

[8] J. Wolter, T. C. Woo, and R. A. Volz, "Gripping position for 3D objects," *Industrial Applications Soc. Meeting*, Oct. 1982.

[9] E. W. Baumann, "Model Based Vision and the MCL Language," *IEEE SMC Conference*, pp. 433-438, Oct. 1981.

[10] E. W. Baumann, "CAD Model Input for Robotic Sensory Systems," *Proc. Autofact IV*, Nov. 1982.

[11] J. D. Cooper, "Why a DoD Standard Programming Language," in *Ada Conference*, Boston, Mass. and Washington, D.C., pp. 1-6, June 17-18 and June 28-29.

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

[13] T. Rentsch, "Object Oriented Programming," *Sigplan Notices*, vol. 17, no. 9, pp. 51-57, Sept. 1982.

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

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

[16] 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.

[17] R. Duda, J. Kremers, and D. Nitzan, "Automatic Part Classification," Fifth Report, NSF Grant G138100X1, SRI Project 4391, Stanford Research Institute, Menlo Park, CA, pp. 7-22, Jan. 1976.

[18] R. Duda and D. Nitzan, "Error Analysis for Automatic Part Recognition," Fifth Report, NSF Grant G138100X1, SRI Project 4391, Stanford Research Institute, Menlo Park, CA, pp. 65-81, Jan. 1976.

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

[20] H. G. Barrow and J. M. Tenenbaum, "Computational Vision," *Artificial Intelligence Center, SRI Int'l.*, vol. 69, no. 5, pp. 572-595, May 1981.

[21] W. A. Perkins, D. B. Juliette, R. Dewar, and J. H. West, "KEYSIGHT: A Computer Vision System for Inspecting Value Spring Assemblies," General Motors Research Laboratories, Research Publication GMR-3601, April 1981.

[22] Stuart J. Kretch, "Robotic Animation," *Proc. Robots* 6, Mar. 1982.

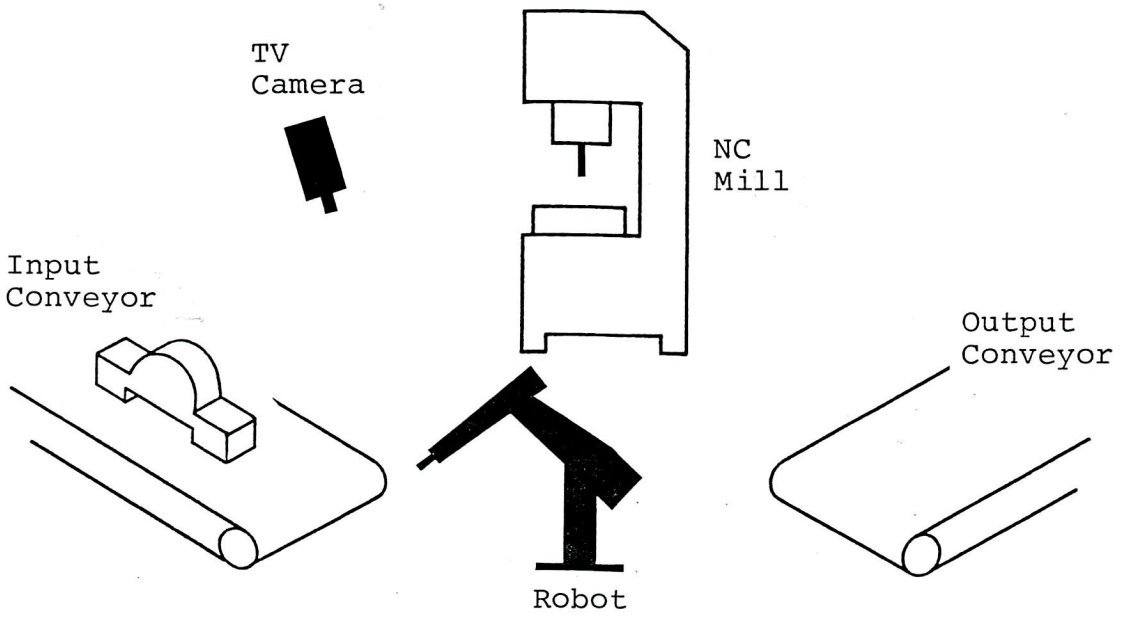

Figure 1. Hypothetical integrated manufacturing cell.

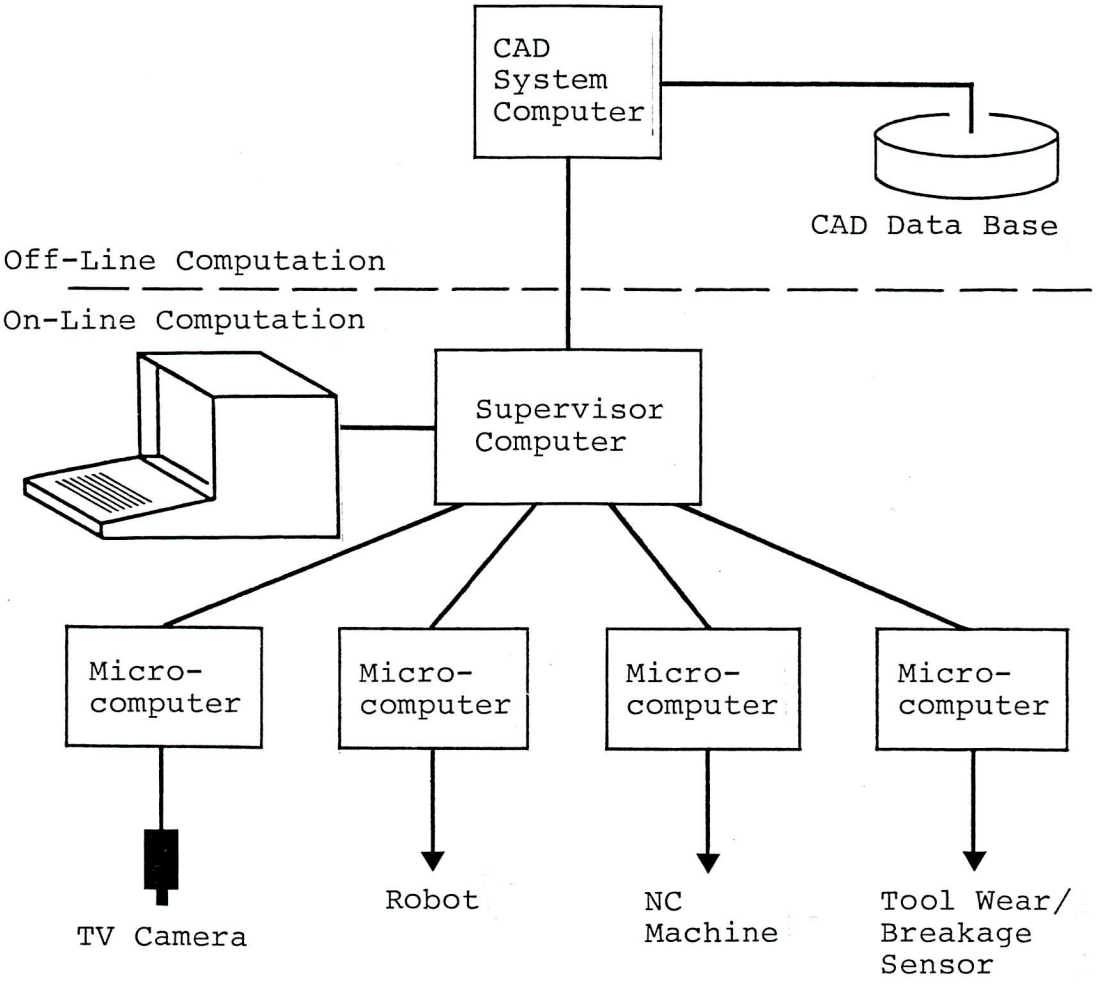

Figure 2. Hierarchical organization of hypothetical manufaturing cell.

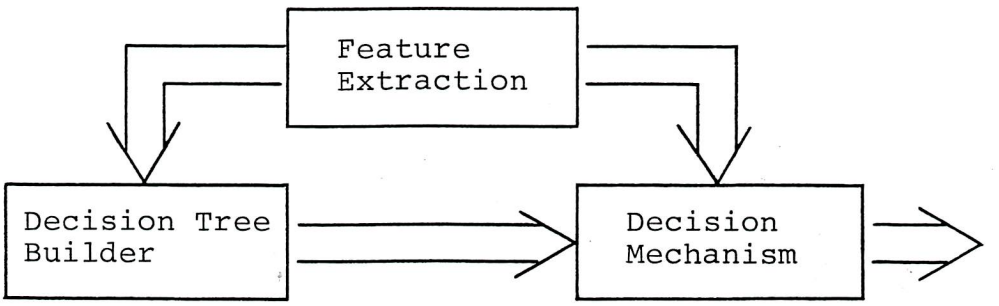

Figure 3. Interrlation of three main vision system modules.

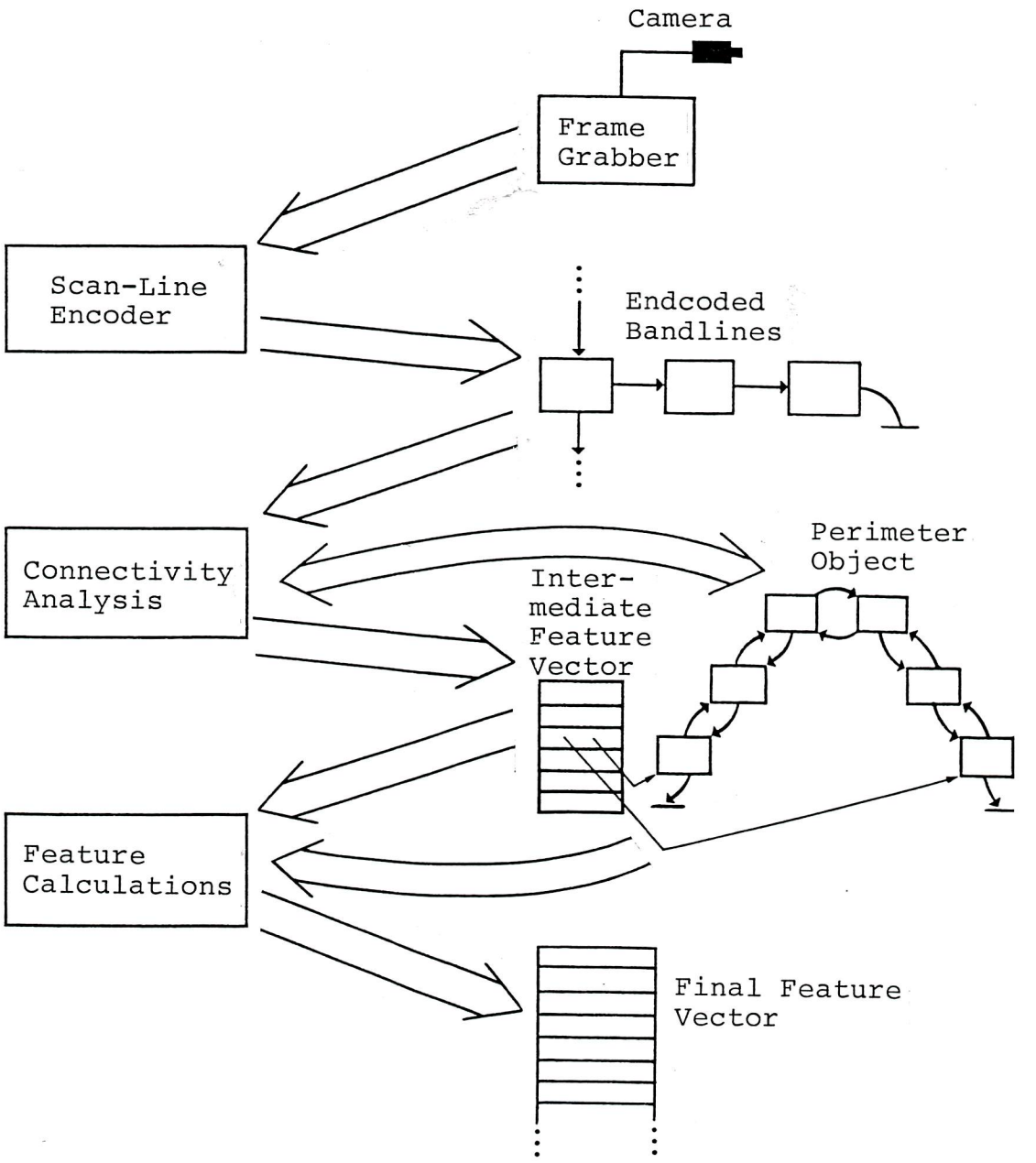

Figure 4. Conventional modularization of vision system.

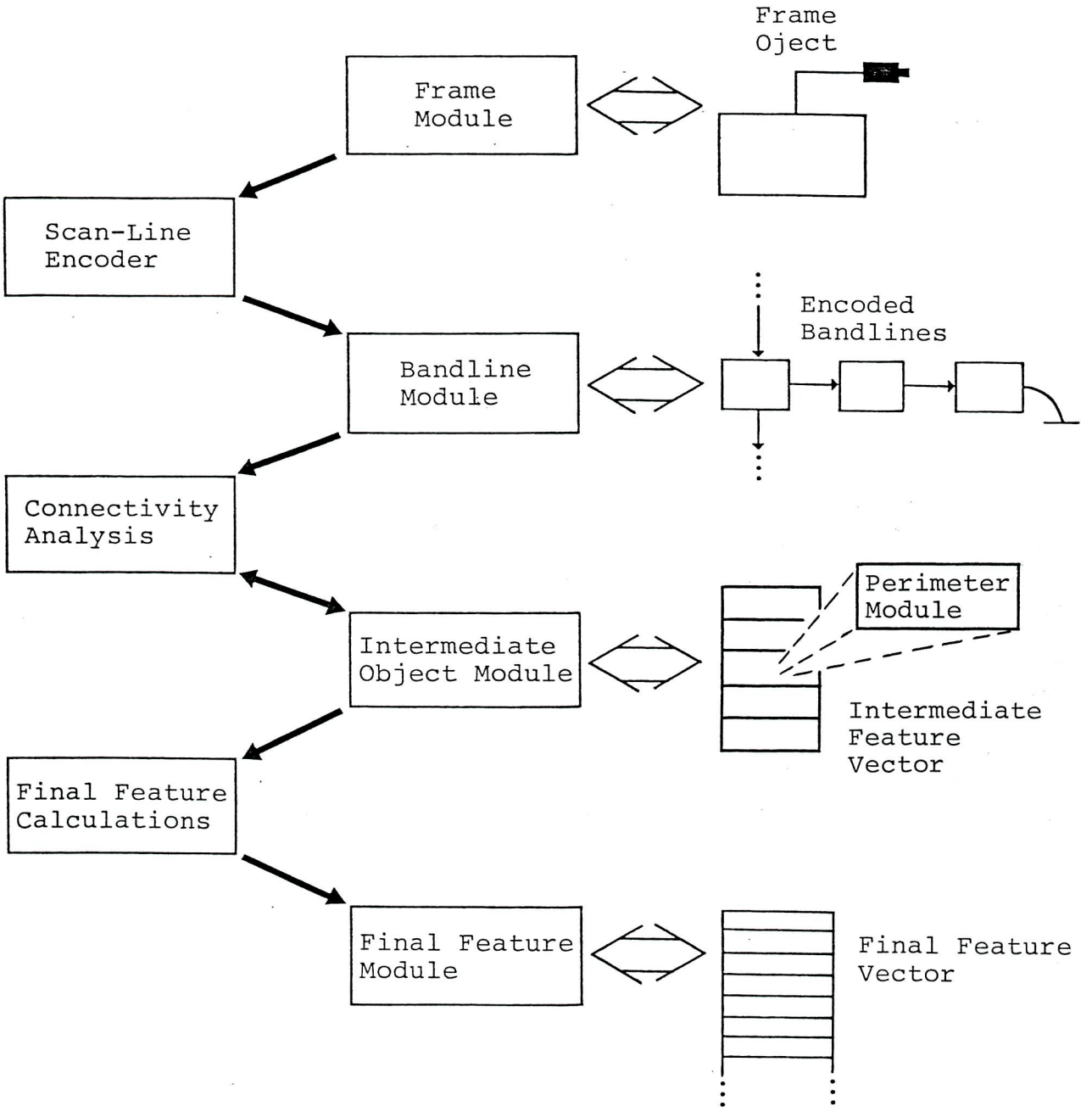

Figure 5. Object-based modularization of vision system.