Comput. Lang. Vol. 7, pp. 1 to 10, 1982
Printed in Great Britain

0096-0551/82/010001-10$03.00/0
Pergamon Press Ltd

INTERACTIVE DEVELOPMENT OF OBJECT HANDLING PROGRAMS

GIUSEPPINA C. GINI* and MARIA L. GINI*
Stanford Artificial Intelligence Laboratory, Stanford University, Stanford, CA, U.S.A.

(*Received* 29 *May* 1979; *revision received* 31 *October* 1980)

Abstract—We describe work on development of a software system for writing and testing programs for a computer controlled manipulator. We examine in particular how the development of working programs is facilitated by the use of an interactive system based on an interpreter. The paper presents the main features of POINTY, the system developed at Stanford Artificial Intelligence Laboratory as a tool for writing assembly programs. The user, interacting with the manipulator, constructs an incremental model of the objects involved in the assembly and develops the corresponding symbolic program.

Special-purpose languages Interactive systems Real time programming Interpreters

1. INTRODUCTION

In this paper we present the basic issues of a software system for developing programs for computer controlled manipulators. We examine in particular how the development of correct programs is facilitated by the use of an interactive system based on an interpreter.

Despite the particular aspects of the investigated application, there appear to be many recurrent themes. The development of tools to aid the user during the writing and debugging phases of his program is recognized as a fundamental requirement for any software system, as well as the importance of a natural interaction between the user and the machine.

The paper presents the main features of POINTY, the system designed and implemented at Stanford Artificial Intelligence Laboratory as a tool for writing AL programs [1, 2]. POINTY was originally developed as a system for describing object models [3, 4], by interacting with the manipulator. Its capabilities quickly went far beyond those initial requirements and it has proved to offer a very reasonable answer to other requirements, so that today it is a complete system for writing and debugging code for manipulators. POINTY was designed with an emphasis on its supportive role as a tool, rather than as a replacement for the user reasoning process. Note that POINTY does not intend to be or to become a system for automatic program generation. Our goal is to propose a system which could aid programmers in constructing manipulator programs in such a way that they will be easier to write, debug and test.

The responsibility of defining object models, of selecting appropriate algorithms is given to the user. The interactions with the external world, the handling of files, the ability of executing source level statements directly is managed by the system, letting the user concentrate himself on the assembly operations.

The choice of the industrial automation domain and the decision to construct an interactive system brought with them several demands. The system has to attract the interest of users. This requires high performance, usefulness and ease of use. The capability of handling an interactive dialogue and extensive human engineering features were from the beginning important aspects in the design of POINTY system. Our experience in using POINTY for real world applications shows that the efforts and the time required to obtain a running program are considerably reduced.

* Present address: Istituto di Elettrotecnica ed Elettronica, Politecnico, Milano, Italy.

Support for this research was provided in part by a fellowship issued by the Italian National Council of Research.

In this paper by the term assembly program is meant the sequence of instructions used to create a mechanical assembly and *not* an "assembly language program" that is used to convert programs into object code. We emphasize this point because we note there could be a confusion between our meaning of assembly program (i.e. used to create a mechanical assembly) and another possible meaning (i.e. a program written in assembly language) which is not meant in this paper.

2. REAL-TIME PROGRAMMING IN HIGH-LEVEL LANGUAGES

Manipulator programming is quite different from other forms of programming. Traditional data processing programs, for instance, rely on a predictable environment, where the only interaction with the external world comes in form of input data; errors in data may be usually corrected without creating emergency situations.

Manipulator programming involves writing and testing real-time programs and interacting with the real world. The unpredictability of the world makes the difference. Running the same program under identical circumstances could produce different results. For instance a small difference in a piece of hardware may cause jams, although the same program works well for other parts [5].

One technique to minimize the effect of randomness of the world is to do extensive testing.

According to [6], real-time debug and test is still a "lost world" compared to the development of other areas of software.

Over the years, high level languages and source language debuggers have been produced, and the testing phase of software has been changed.

Very few of those advantages, however, have affected the world of real-time software debug and test [7].

The typical real-time program is still tested on a computer with no operating system, written in assembler, tested reading absolute machine address contents.

Recent proposals, as [8], to introduce high-level languages in real-time programming seem to guarantee advantages, at least in some applications. Appropriate language structures help in isolating those parts of programs that rely on timing conditions from those parts where time is irrelevant. Source language debuggers many help in debugging data and control structures.

Since errors may have disastrous physical results in the world, it is not ideal to let the complete program run to see what happens. It is more appropriate to approach the problem by developing and testing small portions of the program or even statements. According to this choice, we intend to illustrate here how specialized interactive facilities may transform a high-level programming language into an interactive programming system.

We shall refer to the system for programmable automation designed and developed at Stanford Artificial Intelligence Laboratory [9]. The hardware is made up by two Stanford Scheinman arms, with six degrees of freedom, under control of PDP 11/45. The software is developed and compiled on a DEC PDP 10.

The programming language implemented there, AL [1] allows the user a concise description of the assembly steps, based on world models. A short introduction of AL is appropriate, even though we have not contributed to its definition.

We examine now some of the implicit and explicit assumptions which are part of the methodology developed at Stanford. We believe this will help to suggest the range of problems encountered in manipulator programming and indicate some of the strengths and limitations of the chosen solution.

AL is a real time language for control of real world devices. It is a complete language, and may be used as a general purpose language. Many of the features of ALGOL-like languages have been included in AL. Among these are the block structure and the most widely used control instructions, as well as co-ordination primitives.

Extended data types and the related arithmetic operations have been added because of the recurrent need of dealing with geometrical entities. For instance to describe the

manipulator positions and the locations of objects AL provides frames, which represent a coordinate system in the cartesian space.

Programming a manipulator task usually requires several frames associated with the same object, each of them representing an important position, the grasping position, for instance, or a reference point. When the object position is changed during the assembly, all the frames associated with it must be updated. It would be unfeasible to do that explicitly.

The affix instruction of AL allows to specify that a variable has to be computed from other variables. When we write in a program, for instance "AFFIX BLOCK_1 TO BARM RIGIDLY", where BARM is the name of the arm involved, we want to indicate that a change in the values of BARM or BLOCK_1 requires the updating of the values of the affixed frame. The book-keeping is completely managed by the system.

The variables, their values and the data structure constructed by affixments constitute the AL data model of the world.

The world model is a tree of affixed frames. The nodes represent the physical objects or their subparts, and the arcs the relationships between them. The root of the tree is an implicit object, called STATION, and all the objects which are not subparts of anything else are subparts of STATION. Each arc contains information concerning the relative transformations between the two nodes and specifies whether the relationship is rigid, non rigid or independent.

Since the purpose of any manipulation task is to move objects rather than to get the manipulator in some defined position and orientation, AL allows to describe motions in terms of objects. For instance to move a block to the desired final position we write "AFFIX BLOCK TO BARM RIGIDLY" and "MOVE BLOCK TO FINAL_POS" after having defined BLOCK and FINAL_POS. The affixment relates the positions of BARM and BLOCK, so that changes in the value of BARM will cause corresponding changes in the value of BLOCK; the value of FINAL_POS is then used to compute the hand position that will cause BOX to be at its final place. Motion statements may be enriched by adding modifying clauses, for instance via points, approach and deproach points, controls on force sensors, to perform more sophisticated tasks.

The possibility of using high-level manipulation instructions referring to the objects preserves part of the semantic content that the user puts in his program, and that is obscurated where only the manipulator positions are defined and known.

Those aspects will be discussed below, and a complete assembly program will be illustrated.

3. DEVELOPING AN ASSEMBLY PROGRAM

This section deals with an example chosen with two purposes in mind. First, we want to introduce the reader more deeply into the problems related to the writing of manipulation programs. Object models and their use are easily understood through an example, even though the language cannot be presented here in all its aspects. Second, some of the facilities of the POINTY system will be illustrated, so giving an idea of how much effort and time can be saved using an interactive system.

Our example will involve the determination of the initial and assembled positions of the nut of a valve with respect to a reference point on the base plate on which the parts are located.

The use of a base plate is considered as important. The positions of the component parts will be defined with respect to a corner of the base plate, so that a movement of the main fixture would not require the redefinition of the objects on it.

This feature is not generally available in programming systems for manipulators.

The scene, whose physical set up is due to Shahid Mujtaba, is illustrated in Fig. 1.

Pointing, the method first proposed in the AL language design and developed in [10], has been inserted into POINTY; it involves an interactive system in which the data structures are built by using the manipulator, eventually equipped with a special tool, to point the objects.

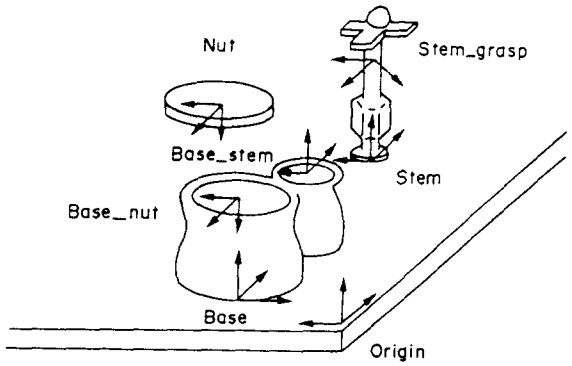

Fig. 1. The assembly station.

The manipulator may be moved around manually or under computer control. It is possible to infer the position of the pointed object from the known joint angles of the manipulator and from the relative position of the pointer with respect to the arm.

A manipulator needs to have six degrees of freedom to be able to position objects in any arbitrary position and orientation. However, while it is easy to position the arm precisely, it is not so easy to obtain the desired orientation so precisely. For this reason multiple readings can be used to determine orientations or a predefined orientation can be specified.

The experience suggests to start using the pointer and defining all the positions not accessible by the hand. We can calibrate the pointer every time, but for the standard pointer we may use the standard calibration, provided by the system. We start defining, as reference point, a frame located at one corner of the main fixture

```
ORIGIN←$ POINTER
```

where the $ sign indicates that we want ORIGIN to have the location of the pointer end and the orientation of STATION.

To define precisely the nut position we remove the pointer from the hand and move the arm manually.

We assume that the orientation of the cap is known to be with the z-axis pointing downwards. We have to determine its horizontal coordinates and the height at which it is to be grasped. The z coordinate can be determined by moving the arm manually to the desired height. Grasping the cap will determine the coordinate of the cap center along a line. A second grasping at 90° to the first will determine the center at a point.
We type in

```
CENTER
```

The Center Command will cause the hand to close until one of the touch sensors triggers. The arm will shift to maintain the touch sensor in contact with the cap until the other touch sensor triggers.

Having grasped the cap, we now read the arm position and assign it to a temporary variable called T1 by means of the command

```
T1 ←↓BARM
```

where the arrow indicates the orientation that we want. We now open the hand to release the cap by the instruction

```
OPEN BHAND TO 3
MOVE BARM TO T1
```

The arm is exactly vertical above the object.

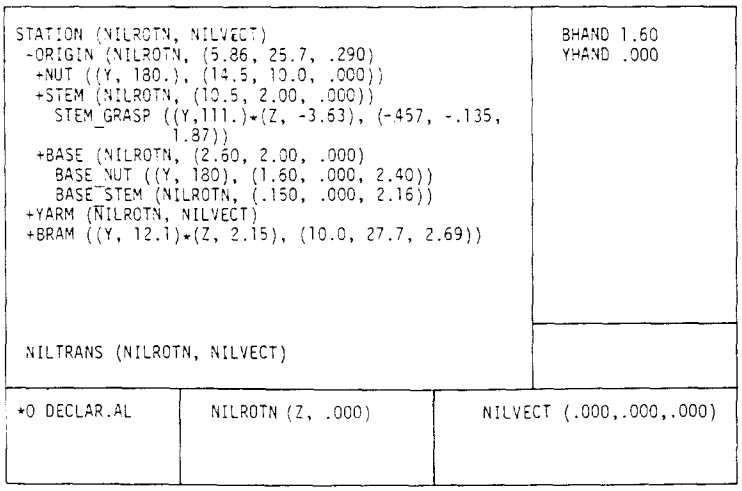

Fig. 2. The display.

Then a rotation by 90° and a new positioning will locate exactly the center of NUT.

DRIVE BJT(6) BY 90
CENTER
NUT ←↓BARM
AFFIX NUT TO ORIGIN NONRIGIDLY

We now move the blue arm with the nut between the fingers to the final position of the nut.

BASE_NUT ←↓BARM
AFFIX BASE_NUT TO BASE

We could have repeated here the same sequence of centering and rotating done before to define the position with higher precision.

We continue the session completing the object model and we save it in symbolic form.

WRITE

The WRITE instruction generates the AL instructions corresponding to the defined world model, and output them on a file (DECLAR.AL for default).

An important tool of an interactive system, a complete display of the situation is maintained on the screen. It gives the user information about the frame tree, the variables and their values, the files used. In Fig. 2 the display is shown as it looks like after the WRITE instruction has been given.

Suppose we want to suspend the session and to save the data model as actually defined. We can save it either in an intermediate form, that can be directly loaded without calling on the parser through the DUMP instruction or in symbolic text form through the WRITE instruction, as seen before.

When the new session is started, after making us sure that the station is in the same configuration as left last time, we can either load a file written in internal form or read a file in text form. Different files in different forms can be read at any moment and the possible contradictions are resolved by the system maintaining what was in memory before reading.

The support of the interactive session is an important requirement. For instance, the history facility of POINTY keeps track of the session, the undo facility allows to back up and undo effects of the previously performed instruction.

Simplified forms of the instructions, default values, avoidance of type declarations and abbreviated forms are examples of human engineering features available in POINTY.

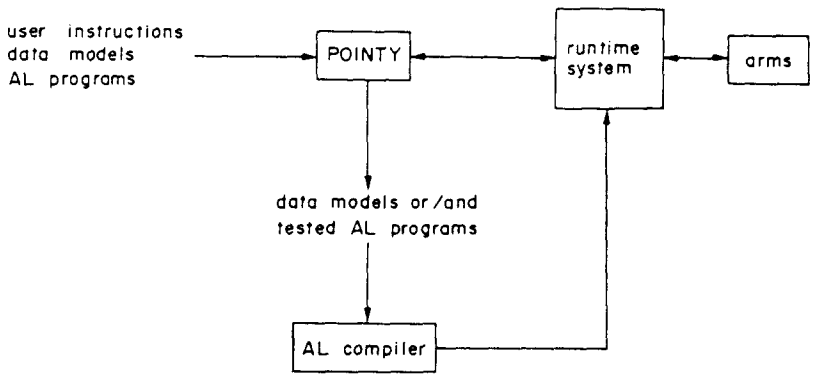

Fig. 3. Organization of the programming system.

Single stepping at the source code level into programs, setting and removing breakpoints, changing values, displaying the symbolic text of the program are available when using POINTY in debug mode.

Editing, debugging, accessing mass memory, executing programs or parts of programs can be interconnected in POINTY, avoiding switching to different separate systems.

4. OVERVIEW OF THE SYSTEM

The problem of producing programs is usually broken down into two interconnected phases. Firstly the program is written and compiled, then it is debugged and tested until the desired task is carried out.

There are many reasons why this does not seem a reasonable approach in the case of manipulator programming, as we have indicated in previous sections. The limitations we have encountered in the traditional approach suggested us the relevance of proposing a different organization of the two phases.

In the first phase the program is written, debugged and tested with the aid of an interactive system, while in the second phase it is compiled by a traditional compiler.

The system we developed is aimed at supplying the user with an interactive, flexible tool to aid him in the first phase of the programming activity. We thought of an interactive system that could facilitate the interaction with the physical world typical of robot programming.

The complete software system developed at Stanford Artificial Intelligence Laboratory is composed by some modules, which carry out the two phases of the programming activity. These modules are shown in Fig. 3 as they are at present.

POINTY allows the user to construct and test assembly programs, using the manipulator as a measuring tool in the cartesian space. The language interpreted is the AL language augmented by some special instructions. Among those the instructions to handle the pointing method, to provide I/O management as well as to supply interactive user facilities. In addition error recovering, specialized editing, a display of the status of the system, an interactive symbolic debugger, undo and simple history facilities are available.

The AL compiler enables the user to compile AL programs to obtain low-level pseudocode that can be executed quickly by the runtime system.

The runtime system interpretes the pseudocode, issuing commands to the device to carry out the task.

While the AL compiler and the runtime system have been developed in recent years and are operational at Stanford Artificial Intelligence Laboratory since 1976 [1, 2, 9], our interest has been concentrated on the design and development of POINTY.

Our approach to the development of programs is quite simple.

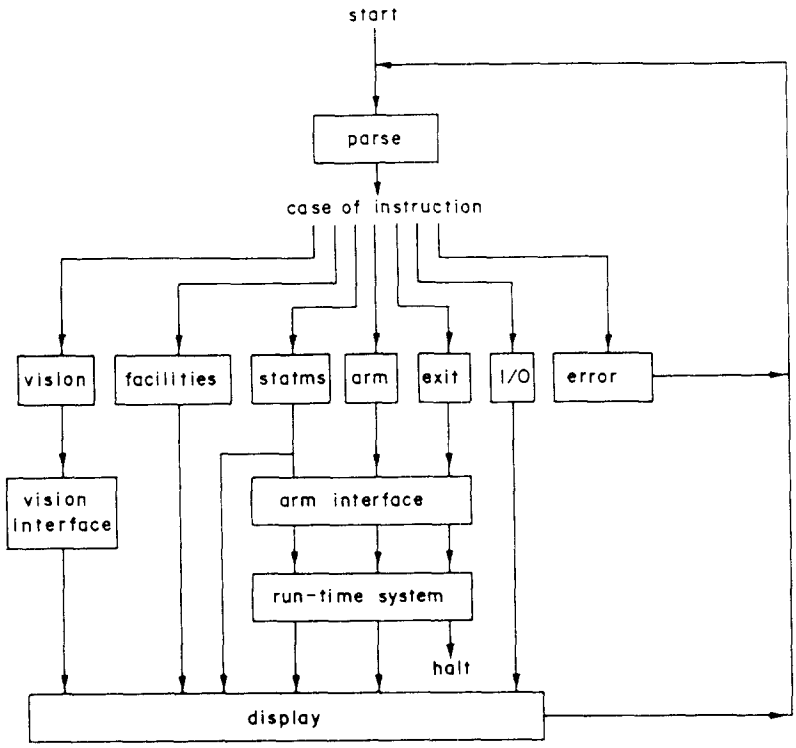

Fig. 4. POINTY organization.

We do not intend to provide an automatic programming system; we simply provide a flexible tool for data models construction and program development. More important, we provide a unified system, in which single modules for single purposes are integrated.

According to [13] "the development of integrated, interactive programming systems, and the methodology for such systems is the major research issue for programming systems and programming methodology today".

The construction of object models is one important feature of the POINTY system. Since describing models is tedious, difficult, time consuming and error prone, special attention has been devoted to the development of sophisticated data collection features. The manipulator may be moved around either manually or under computer control.

From pointed positions and user instructions a model of the objects is constructed. The sequence of AL instructions corresponding to the defined model, ready for insertion in AL programs, is then automatically produced.

POINTY is implemented as an interpreter, whose basic cycle is to read an instruction, execute it, and update the CRT screen with the results while preserving global side-effects in the internal model. Some operating system facilities are supplied for handling files and for managing system facilities.

Besides the standard interpretation, POINTY can operate in debug mode. In this case it reads instructions typed by the user and evaluates them in the context of the place in the program where execution was suspended. The debug mode is particularly useful when programs with compound statements or procedure calls are developed.

The POINTY system is implemented as a set of modules. Since SAIL [11] allows conditional compilation the system can be tailored to the user's needs. For instance, for a user working on a teletype the module that deals with the display generation and updating is not necessary. Analogously an expert user does not need the detailed explanations about the syntax of the language, available as a module.

The organization of the interpreter will be illustrated in the next section.

5. ORGANIZATION OF POINTY

The POINTY system is an extended interpreter which supplies some operating system facilities, as file management, limited editing, powerful symbolic debugging facilities. Its organization is illustrated in Fig. 4.

Every box represents a module; some lower level modules, as the scanner and the symbol table management module, are not indicated, while the error handling system interacts with the parser in a more complex way.

The parser is a top-down parser; it prompts the user for input by means of an "*". The format of the input is free. The user may type one or more instructions per line as well as one instruction on more lines. The semicolon at the end of the line is optional.

The parser is driven by keywords in deciding how to parse the text. It waits for the end of the line and then compares the first token with entries in the keywords table. If there is a match it will follow a sequence of parsing that depends on the token. If no match is found the parser checks if the token is an identifier on the left side of an assignment instruction by looking at the next token. If it is not a back arrow "←", an error message is produced. For arithmetic expressions a simple operator precedence parsing scheme is used.

After the parsing has been completed, the appropriate execution module is called. Whenever an error is encountered during parsing, a syntax error message is given out, while the execution modules give out semantic error messages.

An internal data structure associated with each identifier, containing its type, name, value and, in the case of a frame, its relationships in the tree, is maintained by the symbol table management module, which takes care of the access and operations on the symbol table.

The vision module handles the communications with the vision system [12]. A set of procedures has been defined in POINTY to dialog with the vision system of Machine Intelligence Corporation, and to return position, orientation and other features of the examined objects.

The facilities include editing values of variables, renaming, undoing the last instruction given and its side-effects, asking for help, explaining syntax errors.

The statements module handles assignments of values to every kind of variable, computes arithmetic expressions and object positions from the arm joint angles. It contains facilities for creating frames and relationships between them, as well as for performing operations on the tree with instructions like copy and merge. A full set of Pascal-like control structures, synchonization primitives and parallel processing are available.

The arm module computes the transformations required by movement instructions and sends the appropriate commands to the run time system.

The exit module exits from POINTY, moves the arms to their parking position and closes all the open files.

The Input/Output module contains routines for saving and restoring variables and values in and from a text file of symbolic instructions. This is useful when the user needs to save the AL instructions generated by POINTY or needs to reinitialize the system to a given world state. Other routines manage the loading and the dumping in an internal form which can be directly executed without passing through the parsing.

The error handling module allows interactive correction of errors.

At a lower level the file handling module interacts with the operating system to open, close, write, and read files.

The arithmetic module is called to compute the value of arithmetic expressions. It contains a full set of operations for the basic data types available. It is important to remember that the assembly process involves parts, and parts in elementary terms are bodies that have a location in space and a particular geometry. Description of those concepts is done through the language of mathematics.

All reference frames are considered orthogonal and right handed. Each frame consists of a rotation followed by a displacement vector.

Frames are commonly represented as 4 × 4 matrices where:

the part [1:3, 1:3] is the rotation matrix;
the part [1:3, 4] is the displacement vector;
the last row is constituted by zeroes except the last element, which is one.

This representation is easy to understand and use, since frames may be composed using the ordinary rules for matrix multiplication. Composition of frames, obtained through multiplication on the left, corresponds to a change of reference system.

For instance, if F_A represents the location and orientation of an object A with respect to the work station, and F_{RB} is the location and orientation of an object B with respect to the frame F_A, the location and orientation of this object with respect to the work station is given by

$$F_B = F_A * F_{RB}.$$

The rotation matrix has interesting properties. For instance, its inverse is the same as its transpose.

The rotation of an angle θ about the axis V is expressed by the matrix shown below. Let α, β and γ be the director cosines of V, and let:

$$\begin{aligned} l &= \cos(\alpha) & C &= \cos(\theta) \\ m &= cos(\beta) & S &= sin(\theta) \\ n &= cos(\gamma) \end{aligned}$$

The rotation matrix is:

$$\begin{bmatrix} C + l^2(1-C) & lm(1-C) + nS & ln(1-C) - mS \\ lm(1-C) - nS & C + m^2(1-C) & nm(1-C) + lS \\ ln(1-C) + mS & mn(1-C) - lS & C + n^2(1-C) \end{bmatrix}.$$

Algorithms to deal with matrices constitute a large part of the arithmetics module.

The manipulator interface provides the communication between the user program and the manipulator. It allows communication with the run-time system of the PDP 11, to transfer the arm joints values to the PDP 10 and to send commands for movements to the arms.

The run-time system is not a POINTY module; it is the system resident on PDP 11 devoted to the arm servoing and to the execution of the pseudocode obtained with the compilation of AL programs, as illustrated in [5].

At the end, the display module updates the screen of the user's terminal to reflect the current state of the model being defined. As soon as POINTY is loaded, the display is generated on the screen and it is continuously updated after each instruction.

6. CONCLUSIONS

We presented a versatile "aid-to-programming" system to be used as a tool for developing manipulator programs.

We introduced some considerations about the opportunity of employing high-level manipulator languages based on world models. After having presented AL, one of such languages, we introduced POINTY, an interactive system designed for constructing AL world models as well as for writing and testing parts of AL programs.

The implementation of POINTY has been tested, demonstrating that the specification of object can be easily obtained and the writing of AL programs can be significantly simplified as well.

Since the presented system is based on the interaction through the manipulator with the physical world it may suggest a new philosophy in robot programming, that will combine some aspects of the traditional teaching-by-guiding approach with the use of advanced software facilities.

The association of those two aspects in a unique system discloses, in our opinion, a new interesting direction in manipulator programming, both for experimental systems and for industrial applications.

Acknowledgements—We would like to express our thanks to Tom Binford and all the other members of the Hand-Eye group; in particular we thank Shahid Mujtaba, who contributed to the implementation of POINTY and now maintains the system.

REFERENCES

1. R. Finkel *et al.*, An overview of AL, a programming system for automation, *Proc. 4th IJCAI*. Tbilisi, USSR (1975).
2. M. S. Mujtaba and R. Goldman, *AL Users' Manual*. Stanford Artificial Intelligence Laboratory, Stanford, CA (1978).
3. G. Gini and M. Gini, Object description with a manipulator, *Ind. robot* **5**, 32–35 (1978).
4. G. Gini and M. Gini, POINTY: a philosophy in robot programming. In *Information Control Problems in Manufacturing Technology* (Edited by Rembold *et al.*), pp. 173–181. Pergamon Press, Oxford (1980).
5. R. Finkel, Constructing and debugging manipulator programs, Ph.D. Thesis, Stanford Artificial Intelligence Laboratory, Memo AIM-284. Stanford, CA (1976).
6. R. L. Glass, Real-time: the "lost-world" of software debugging and testing, *Commun. ACM* **23**, 264–271. (1980).
7. S. Lauesen, Debugging techniques, *Software Pract. Exper.* **9**, 51–63 (1979).
8. N. Wirth, Toward a discipline of real-time programming, *Commun. ACM* **20**, 577–583 (1977).
9. T. O. Binford *et al.*, Exploratory study of computer integrated assembly systems, Progress Report 4, Stanford Artificial Intelligence Laboratory Memo AIM-285.4. Stanford, CA (1977).
10. D. D. Grossman and R. H. Taylor, Interactive generation of object models with a manipulator, *IEEE Trans Syst. Man Cybernet.* **SMC-8**, 667–679 (1978).
11. J. F. Reiser (ed.), SAIL, Stanford Artificial Intelligence Laboratory Memo AIM-289, Stanford, CA (1976).
12. C. A. Rosen, Machine vision and robotics: industrial requirements, *International Symposium on Computer Vision and Sensor-based robots*. General Motors Research Laboratories. Warren, MI (1978).
13. E. Sandewall, Programming in an interactive environment: the 'LISP' experience, *Comput. Surv.* **10**, N. 1, March 1978.

About the Author—GIUSEPPINA GINI received her doctoral degree in physics from the State University of Milano, Italy in 1972. She was research fellow during the next four years at the Politecnico of Milano in the Artificial Intelligence Project. Then, she spent more than two years at the Artificial Intelligence Laboratory of the Stanford University. At present she is research associate at the Politecnico of Milano. She has published several papers in the area of programming languages for Artificial Intelligence, robot programming and program construction.

About the Author—MARIA GINI received her doctoral degree in physics from the State University of Milano, Italy in 1972. She worked as a research fellow from 1972 at the Politecnico of Milano. During this period she spent more than two years at the Artificial Intelligence Laboratory of the Stanford University. At present she is research associate at the Politecnico of Milano. Her research interests are in the field of Artificial Intelligence and its applications to robotics. She is the author of several publications on those subjects.