Dealing with World-Model-Based Programs

GIUSEPPINA C. GINI AND MARIA L. GINI
Stanford Artificial Intelligence Laboratory

We introduce POINTY, an interactive system for constructing world-model-based programs for robots. POINTY combines an interactive programming environment with the teaching-by-guiding methodology that has been successful in industrial robotics. Owing to its ability to control robots in real time, and to interact with the user, POINTY provides a friendly and powerful programming environment for robot applications. In the past few years, POINTY has been in use at Stanford to write, test, and debug various robot programs.

Categories and Subject Descriptors: D.1 [**Software**]: Programming Techniques; D.2.6 [**Software Engineering**]: Programming Environments; I.2.9 [**Artificial Intelligence**]: Robotics; I.2.5 [**Artificial Intelligence**]: Programming Languages and Software; J.6 [**Computer Applications**]: Computer-Aided Engineering

General Terms: Design, Performance

Additional Key Words and Phrases: World modeling, teaching-by-guiding, testing and debugging, software development, interactive systems, manipulators

1. INTRODUCTION

In this paper, we present a methodology for programming robots. We discuss issues in implementing that methodology for a computer-controlled manipulator. We report our experience in developing and using such a system.

Today, robots are operating on a wide variety of tasks such as object handling, painting, and welding. Even though assembly is still considered a difficult area, more and more robots are used in manufacturing to perform assembly tasks [18]. New areas outside manufacturing, such as exploration of unknown environments and medical applications, are being considered.

All these new developments raise crucial problems. Programming robots, which is very easy when tasks are simple, pick up and place for instance, becomes an important issue. Sensors provide a huge amount of potentially useful data that have to be interpreted to be of any use. Computer programming, which was nonexistent in the early stages of robotics, becomes one of the central issues.

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

Authors' current addresses: G. C. Gini, Dipartimento di Elettronica, Politecnico, Milan, Italy; M. L. Gini, Dept. Computer Science, University of Minnesota, 207 Church St., SE, Minneapolis, MN 55455.

Programming a computer-controlled robot is really different than programming a computer. Robot programs run in a world which is incompletely known and imperfectly modeled, requiring strategies to detect and prevent potential catastrophes, such as collisions, and to recover from errors. Many actions are irreversible. After the arm has crashed there is no way to undo the action that produced the crash. Actions are not exactly reproducible, making it difficult to detect the causes of errors and to correct them [9].

Robot tasks should be programmed in a simple and natural way, without extensive user training. This goal has been achieved with the industrial robots, programmed by guiding the arm through the motions of the task. The sequence of the relevant positions is simply stored for further executions. This methodology, known as teaching-by-guiding, has been successful for tasks where only simple operations or few positions are required. When complex assemblies have to be performed this method is not practical. Its main drawback is that it does not allow any adjustment of robot movement so as to take sensor data into account. In the case where the position of parts in the assembly station is even slightly modified, the teaching has to be repeated in its entirety.

Writing programs for robots is not as easy as one would think [14, 15]. Manipulation and assembly tasks are hard to program mainly because it is difficult to visualize positions and orientations in three-dimensional space. Interactions with sensors [23, 28] are hard to express because it is difficult to figure out the directions and values of forces. The design and development of POINTY has been motivated by the desire to combine the ease of the teaching-by-guiding method with the power of an interactive programming system based on a high-level language.

For users, POINTY is an executor of input commands. Each command is an instruction or a program written in an Algol-like language. Instructions include conventional Algol-like statements such as assignments, conditional expressions, and loops, in addition to manipulator control and synchronization instructions. The language interpreted by POINTY is AL, an Assembly Language designed in the last decade at the Stanford Artificial Intelligence Laboratory.

POINTY reads and parses each instruction, interprets it, and turns to the next instruction, interacting with the physical devices according to user requirements. All the instructions, including the ones controlling the robot and the vision system are executed in real time, giving users a complete and immediate control over the robotics system.

The ability to interact with the robot arms is very important during the development of programs. Users can execute one instruction at a time and construct step-by-step a program that is completely satisfactory to them. This tends to suggest a bottom-up development of programs, which seems to be quite natural in robotics. One should realize that most of the time the algorithms used in robot programs are simple. What is difficult is to make the robot do what one wants. A bottom-up development allows an immediate testing of the different parts of the program before connecting them.

Another important aspect of POINTY is its ability to generate symbolic instructions corresponding to the world model defined by the users during an interactive session.

The results of a POINTY session are a symbolic program and the corresponding symbolic world model. Traditional teaching-by-guiding systems produce as output a stored sequence of joint coordinates instead of a symbolic program. More advanced systems can generate values of robot positions in Cartesian space; they do not generate symbolic instructions.

Two points of our system should be noted. First, POINTY provides a complete and unified programming system for a real-time environment. Second, it provides software tools in an application area where software is often very poor.

Section 2 discusses the problem encountered in robot programming. The rest of the paper is devoted to the presentation of the main features of POINTY. We show how POINTY makes it easier to construct world models and robot programs. Section 3 presents an overview of the POINTY design criteria and compares it with analogous systems. A programming example is shown in Section 4.

2. SOFTWARE FOR ASSEMBLY ROBOTS

Since computer-controlled manipulators have been introduced as a general-purpose mechanism for industrial automation, the methodology of controlling and programming them for new tasks has been the subject of a great deal of development [1, 3, 13, 20].

Two completely different approaches to robot programming have been considered in the past. On the one hand, within the artificial intelligence community, a lot of research has been done on plan formation systems to provide robots with autonomous reasoning capabilities. None of these systems have been used to control a real robot, with the exception of STRIPS at SRI [4].

One of the reasons why plan formation systems have not been used with real robots is that in manufacturing applications the sequence of actions is known in advance. There is no need to do any planning, and, furthermore, planning requires a lot of computer power. But when the environment changes in an unpredictable way the ability to do some reasoning becomes essential.

Moreover, complex tasks requiring more than one robot working at the same time are difficult to program. Parallel actions are hard to express. Time constraints and optimization in the use of available resources require a considerable programming effort. None of the robot languages offer reasonable primitives for parallel actions, while some plan formation systems can deal with multiple agents [12, 24], resource sharing, parallel planning [32], and planning in time [31].

On the other hand, the need to control industrial robots has pushed the development of simple but effective methods for robot programming [8, 16, 19]. Complex systems have been designed over the years to cope with increasing demands [27, 30]. None of them yet require the reasoning capabilities provided by artificial intelligence.

Programming a robot requires writing a computer program, debugging, and testing it until the task is carried out in a correct way. The interaction with the physical world makes this task much more complex than it first appears.

The physical world is much more unpredictable than the computer world, and any program, even if it is correct, may fail to achieve the intended goal. For instance, a little oil spot on a piece of hardware may cause the robot hand to

drop it, although the same grasping force works well for other, identical pieces. One can recover from failures caused by arm errors, like dropping or failing to grasp an object, only by using ad hoc error recovery procedures. In writing and debugging manipulation programs, users must depend on their experience, intuition, and common sense to decide what errors to watch for.

A model of the surrounding world has to be inserted in the robot program. Knowing whether the amount of information provided is sufficient for the purposes of the program is not a trivial task. World models reflect real objects in some of their aspects only, depending on the choices the programmer makes. During execution of the program, users may discover that the model is incomplete in a crucial way, and that some other features of the physical world should have been known.

Another characteristic of the physical world is that many actions are irreversible. Actions such as insertions of parts cannot be reversed automatically, since the forces required during the backward motion cannot be easily derived from the forces applied during the insertion.

Last, but not least, the testing cycle is complicated by the necessity of performing a longer sequence of actions. In order to fix a known error it is necessary not only to modify the source code, to resubmit the program to the compiler, and to load it again, but the physical environment must also be reinitialized to the starting configuration, if it is possible to do so. The failure may not be exactly reproducible, so it may not be feasible to test the new code easily.

The implementation of software tools, as symbolic debuggers [6, 17], file systems, and editors, may be seen as an important step in the direction of reducing the efforts required to obtain running programs [26].

The need for a complete programming system has been envisaged and discussed in various other papers also [14, 22]. The ideal system should have a simple syntax and semantics, and the following desirable features:

(1) Data structures should accommodate complex symbolic information as well as primitive types. Data structures should be free to grow in unrestricted ways, and storage declarations should be optional to the user.
(2) Strong I/O and file manipulation facilities must be included. It should be possible to save programs and data both in symbolic or internal form; it should be easy to access files from programs. Performance data and log files are some of the very important capabilities required.
(3) The control of many parallel events should be easily available.
(4) There should be a language interpreter for software development and debugging. Moreover, the language should also have a compiler for use in production.
(5) A set of utilities (language-oriented editor, debugging facilities, source program formatter, a self-modifying display, and so on) should be made available.

A few high-level manipulation languages have been designed in recent years, among them are AL [2, 17], AML [30], AUTOPASS [13], RAPT [21], VAL II [27]. (A good classification and comparison of many of them can be found in [3]). These languages reduce the amount of detail that programmers need to

consider, letting them concentrate on the most important aspects of the task at hand.

Even the most advanced languages suffer from many drawbacks, making it painful to write complex programs. Errors are difficult to identify because of their unpredictability. The same program can work well hundreds of times and then stop because of a minimal variation in the size of one part [9]. Even Ada, used as a robot programming language [7], does not offer a much better solution to this problem.

In this paper we refer to the experimental system for programmable automation, designed and developed at the Stanford Artificial Intelligence Laboratory [1, 17]. At the time that POINTY was implemented the system was composed of two Stanford Scheinman arms, each with six degrees of freedom, which allowed them to be positioned in any arbitrary position and orientation; a vision machine from the Machine Intelligence Corporation was also part of the system. The software for controlling and programming the arms was developed and compiled on a PDP 10 and run on a devoted PDP 11/45.

We now examine some of the assumptions, either explicit or implicit, that are part of the programming methodology developed at Stanford. We believe this will help in understanding the problems encountered in robot programming, and indicate some of the strengths and limitations of our work.

AL is a real-time language for control of real-world devices. It is a complete programming language, with most of the features of Algol-like languages and coordination primitives. It was originally designed as a compiled language. To be more precise, any AL program is translated by the AL compiler into an intermediate p-code that is later loaded on a different computer and executed. POINTY has shown that a fully interpreted language gives more flexibility and a better interaction with the robot. The compiler is, however, useful for repeated execution of programs. We believe that this programming environment, offering both an interpreter and a compiler—quite unique in robotics—has advantages over other systems.

Special data types and the related arithmetics are an important part of AL, because of the recurrent need of dealing with physical entities.

To describe manipulator positions and objects, AL provides frames which represent a coordinate system in Cartesian space. The six values of a frame represent the six degrees of freedom of a rigid body in space. These values are needed by the manipulator to compute the values of its joints.

An important concept in AL is the world model. The world model is a tree of affixed frames. Each frame constitutes a reference system in Cartesian space and represents an interesting feature of the object such as its grasping point or its position. The affixment relation specifies the dependencies between the different frames in order to avoid the bookkeeping involved in modifying their values each time one of them is moved.

An example can better explain the world model required by AL programs. The scene illustrated in Figure 1 can be described by the set of AL instructions given in Figure 2.

A schematic description of the world model is illustrated in Figure 3. The arcs are marked according to the type of affixment relation. The term "rigidly"

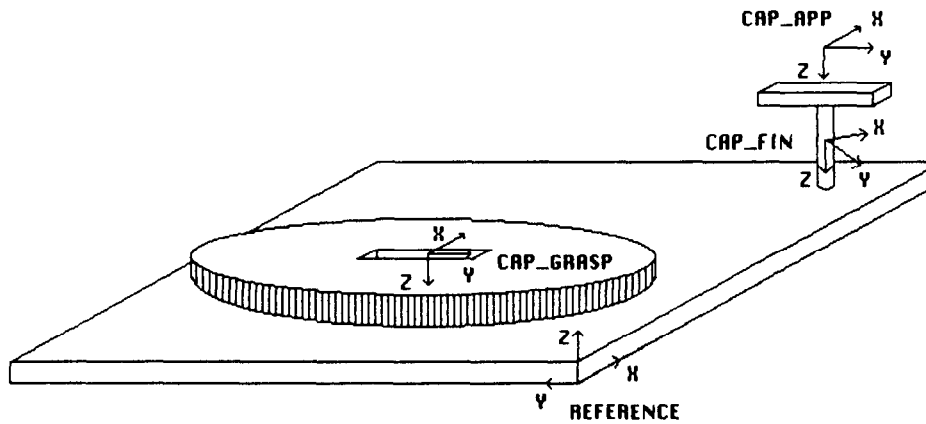

Fig. 1. The assembly station.

```
FRAME REFERENCE;
REFERENCE :- FRAME (NILROTN, VECTOR (5.38,34.1,.216) * INCHES);
FRAME CAP_GRASP;
AFFIX CAP_GRASP TO REFERENCE AT TRANS
      (ROT (YHAT, 180.*DEGREES) * ROT (ZHAT, -179.3 * DEGREES),
       VECTOR (2.67,9.22,1.08) * INCHES) NONRIGIDLY;
FRAME CAP_APP;
AFFIX CAP_APP TO REFERENCE AT TRANS
      (ROT (YHAT, 180.*DEGREES) * ROT (ZHAT, -180.* DEGREES),
       VECTOR (10.2,3.71,3.11) * INCHES) NONRIGIDLY;
FRAME CAP_FIN;
AFFIX CAP_FIN TO CAP_APP AT TRANS
      (ROT (ZHAT, 45.0*DEGREES), 3*ZHAT) RIGIDLY;
SCALAR WIDTH;
WIDTH :- 2.4;
```

Fig. 2. The AL world model.

indicates a symmetric relation, while "nonrigidly" indicates a one-directional relation.

Once the positions of the parts have been defined, the program is a sequence of movements describing the movements of objects (or frames) rather than those of robots.

There are some advantages in using world models, even in the limited form offered by AL.

–First, the program can make reference to objects rather than to manipulator positions. This preserves the program part of the semantic content and makes it easier to use the same program for a different robot.

–Second, the same assembly program can be used for different world models, corresponding to different configurations of the same parts in the assembly station. Only the positions of the objects have to be recomputed, and no changes

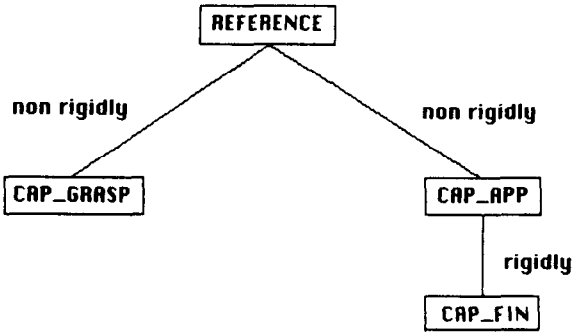

Fig. 3. The frame tree.

should be made to the assembly algorithms. When positions of objects are defined as relative to one corner of the fixture, only the position of the fixture needs to be updated each time it is moved.

3. INTERACTIVE WRITING OF MANIPULATOR PROGRAMS

The integration of teaching-by-guiding methods with symbolic programming has been attempted in industrial robot systems as a way to reduce the time required to produce working programs.

In 1975, an industrial robot, the SIGMA of the Olivetti Company, was provided for the first time with a programming language [25] and an interconnected joystick. Users were able to write programs, teach positions, and insert them into the program. This solution is now common in the more recent industrial robots, as in Unimation's PUMA [27]. Only data about arm positions are collected from the arm, and are then stored with a symbolic name which can be used in the program.

Problems become different when the level of the language increases. In the philosophy of a language such as AL one does not teach positions but world models. A programming module able to teach the robot such data has been developed by Grossman and Taylor [10], and special hardware has been proposed by Hasegawa and Inoue in [11].

The implementation of POINTY that we present here is a completely new one. It is interesting to note that the idea of Grossman and Taylor [10, 29] was to make it easier to code world models for AL programs. They thought that writing the program was easy in AL and that the really difficult part was the description of the world model. This is certainly true, but does not solve every problem.

Once the world model has been constructed, it has to be tested with the complete program. The program usually requires many refinements. The most useful aspect of POINTY is the fact that it provides an interactive system for writing and executing AL programs. In fact, the choice of implementing POINTY as an interactive AL has proved to be the key to the success of POINTY.

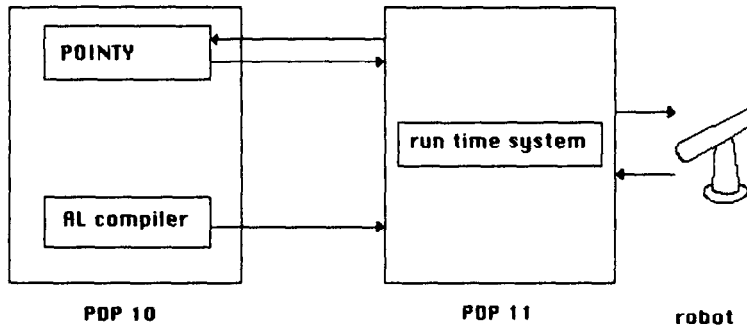

Fig. 4. The organization of the software system.

The system we designed and implemented is part of a complete programming system organized in three main modules, as illustrated in Figure 4:

-the program development system, POINTY, supplies the user with interactive facilities to specify and execute assembly tasks; it interacts with the run-time system that executes the instructions;
-the AL compiler enables the user to compile AL programs in order to obtain a low-level pseudocode that can be executed rapidly;
-the run-time system interprets the pseudocode, sending commands to the manipulators to carry out the task.

While the compiler and run-time were available at the time our research started [5, 29], our interest has concentrated on the program development system, POINTY.

POINTY has a very wide range of applications. It does not intend to impose a specific philosophy of programming, although bottom-up and modular development of programs is strongly facilitated. Users start with one simple operation and try it until they are completely satisfied. By combining different operations in procedures or blocks, they can construct larger and larger modules.

The robot arms may be moved around manually or under computer control. Positions and orientations of objects can be defined easily in terms of arm positions.

Some points should be noted: POINTY operates in real-time, supports cooperating processes, and is able to control two arms and to interact with the vision system of the Machine Intelligence Corporation.

The system is distributed on two computers, each one devoted to a different task and communicating with the other. POINTY is implemented on a time-shared PDP 10 interfaced with a PDP 11/45. Arithmetics and computation are performed on the PDP 10, while the PDP 11 is responsible for the control of the manipulators. Although one objective of industrial robot systems is to minimize the need for large capacities of computing power, the philosophy of our programming system requires large computer resources. This problem is not relevant in our discussion here, since inexpensive and powerful hardware should be available in the near future.

The language interpreted by POINTY is the same as AL, although some specific instructions have been added to support the interactive session. This choice was critical from the very beginning. If users can write the same program for POINTY and AL, they are strongly motivated to use POINTY for program development, and to then call the AL compiler only when the program is stable and completely tested. The use of the POINTY system has shown that users want to work with the same language. The compatibility between POINTY and AL has been one strong point in favor of an extensive use of the POINTY system.

There are two modes of operation in POINTY: the interpreter mode and the debug mode. The first corresponds to the execution of one statement at a time, the second allows breakpoints and single-stepping at the POINTY instruction level.

When in debug mode, users can set and remove breakpoints, restart the program or abort it, and do single-stepping at the source code level. The symbolic text of the program under execution can be displayed completely or partially, together with the positions of breakpoints. Different types of single-stepping allow users to step through the procedures and compound statements or to execute them as a single instruction.

In debug mode, POINTY reads instructions typed by the user and evaluates them in the context of the place in the program where execution was suspended. The evaluation is performed just as if the users had inserted an extra statement into the original program at the point where execution was suspended. Users may ask to evaluate any POINTY expression whose evaluation would be legal at the point at which the execution of the program was suspended.

For ease of presentation, we can classify instructions into the following classes:

(i) *Operations on variables and data types.* Besides the classical Algol-like instructions, there are arithmetic operations on scalars, vectors, rotations, transformations, and frames, the construction of trees and the deletion of variables. Values may be assigned as being absolute or relative to a reference system. Even though the language is typed, declarations may be omitted, because the system supplies implicit declarations, when they are not present, according to the syntax of the language. Both macros and procedures can be defined, saved, and recalled when needed.

(ii) *Interactions with real-time devices.* The full set of instructions to control the robot arms and to read their positions is available. Frames, vectors, and rotations can be defined using the robot as a measuring tool in three-dimensional space. The availability of computer-controlled movements of the robot arms allows execution of programs and checking their correctness. Different processes can be defined and executed in parallel.

(iii) *User interface and utility packages.* A number of features make the POINTY system easy to use. I/O operations allow reading files with AL instructions in symbolic or binary form, generating instructions corresponding to the internal data model, and saving them on mass storage files. Renaming of variables and editing of values are provided in the language. Typing has been minimized by use of defaults and abbreviated forms. An extensive use of line editing capabilities allows easy modification or correction of instructions. Help can be requested at any moment. The state of the program is continuously displayed on

the screen. To save the complete session, or parts of it, the terminal output can be saved and the data model generated in the form of symbolic instructions. Files with procedures can be constructed and called when needed.

4. USING THE SYSTEM

We now illustrate a complete POINTY session. Since it is difficult to describe in a short example all the features of the system, we illustrate a typical sequence of actions taken to write a world model for an assembly program.

Suppose we have the working station arranged as in Figure 1, and we want to write an AL program to control the robot in assembling the parts shown there. We describe step-by-step how to use POINTY for generating the symbolic model illustrated in Figure 2.

We start by defining a reference point, located at one corner of the main fixture. We later define the other frames as relative to that reference point. In this way any possible movement of the fixture will require only the appropriate modification of the reference point. We manually move the hand of the chosen arm to that point, and we type:

REFERENCE := $ BARM

BARM is the variable that represents one the Stanford arms. The system can recognize the type of the variables, depending on the way they are used. This saves some typing, which is important while working with an interactive system. In this case, REFERENCE is recognized as a frame. The $ sign indicates that we want REFERENCE to have the STATION orientation. STATION is a frame that constitutes the main reference system. It is located at one corner of the working area, with the z axis pointing upwards. The position of STATION is a three-dimensional vector (called NILVECT) with all the components equal to zero; its rotation part is a null rotation (called NILROTN).

REFERENCE is a frame whose vector part is the same as the position of the robot. The rotation part is the same as STATION.

We move the hand manually to the cap to define its grasping position, and we type:

CAP_GRASP := ↓ BARM

where the arrow indicates that we want a frame with the z axis oriented downwards, independently of the arm orientation. In that way, when we move the arm to the cap we will have the arm exactly perpendicular to the working plane. It should be noted that manual positioning makes it almost impossible to obtain precisely the desired orientation. This is a typical case in which the specification of the frame CAP_GRASP is underdetermined. We want to specify the position of the hand only and any orientation normal to the working plane. Since in AL each frame is completely specified, we introduced this feature in POINTY to allow more flexibility.

AFFIX CAP_GRASP TO REFERENCE NONRIGIDLY

affixes the two frames nonrigidly. A nonrigid affixment indicates a nonsymmetric relationship between the two parts. Moving REFERENCE will move CAP_

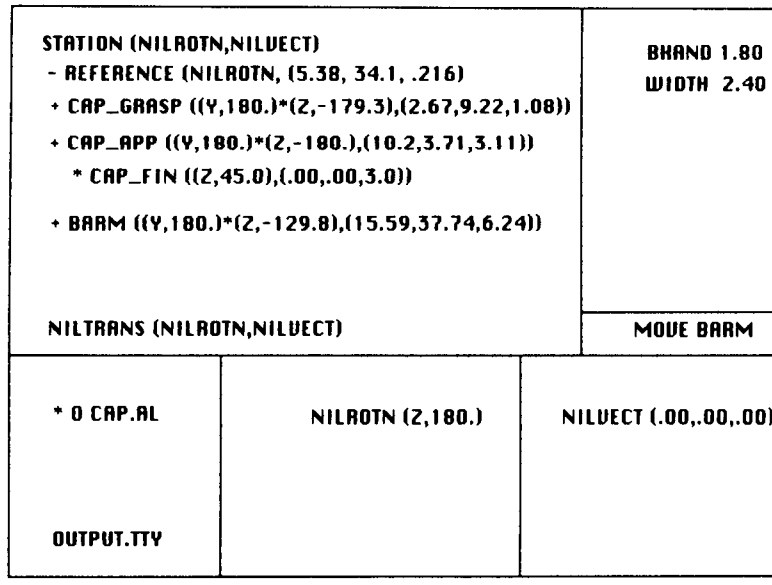

Fig. 5. The display.

GRASP. The opposite is not true. This takes care of updating the value of CAP_GRASP in the case where REFERENCE is updated, but will not change REFERENCE when we move CAP_GRASP to its final destination.

We now move the arm with the cap between the fingers to the position required for insertion, and we define a new frame that will be used to approach the final position:

CAP_APP := ↓ BARM

The final position of the cap is obtained with a rotation of 45 degrees about the z axis. The instruction:

AFFIX CAP_FIN TO CAP_APP AT TRANS (ROT (ZHAT 45), 3*ZHAT)

defines a new frame, CAP_FIN, using the given transformation as the relative position of CAPFIN with respect to CAP_APP, and affixes the two frames.

The last affixment completes the world model.

AFFIX CAP_APP TO REFERENCE NONRIGIDLY

The hand opening can be used to identify the width of the part to be grasped.

WIDTH := BHAND + .6

The world model may be saved on a file in the form of AL instructions. The instruction

WRITE CAP.AL

will do that for us by writing the AL instructions corresponding to the object model defined on the file CAP.AL. These instructions have been illustrated in Figure 2.

In the case where we intend to restart the POINTY session in the future, we may instead save the model constructed in an internal form through the DUMP_VARIABLES instruction. This allows us to reload the model more rapidly, without calling the parser.

A complete display of the situation is maintained on the screen, providing users with information about the frame tree, the variables and their values, and the files used. In Figure 5 the display at the end of the session is shown.

After the construction of the object model we continue writing and testing the program to perform the assembly operations. The complete description of these phases is beyond the aim of this paper. Readers may find an example and some more details in [17].

5. CONCLUSIONS

Our aim in this paper has been to present an interactive system to be used for developing real-time programs for the control of robots.

We have introduced some considerations about the need to employ high-level languages based on world models. After having presented one such language, AL, we presented POINTY, an interactive system designed for constructing AL world models, as well as for writing and testing AL programs. POINTY has been implemented and tested.

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

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

ACKNOWLEDGMENTS

The authors are indebted to Tom Binford for his many valuable suggestions, to Shahid Mujtaba for his help in the implementation of the system, and to all the other members of the Stanford Hand–Eye group for their long and useful cooperation. Special thanks go to the referees for their useful comments on previous versions of this paper.

REFERENCES

1. BINFORD, T. O., ET AL. Exploratory study of computer integrated assembly systems. Progress Rep. 4, Stanford Artificial Intelligence Lab. Memo AIM-285.4, Stanford, Calif., June 1977.
2. BINFORD, T. The AL langauge for an intelligent robot. In *Languages et Methodes de programmation des robots industriels*, IRIA Press, Paris, 1979, 73–88.
3. BONNER, S., AND SHIN, K. A comparative study of robot languages. *Comput. Mag. 15*, 12 (1982), 82–96.

4. FIKES, R. E., AND NILSSON, N. J. STRIPS: A new approach to the application of theorem proving to problem solving. *Artif. Intell. 2* (1971), 189–208.
5. FINKEL, R., ET AL. Overview of AL, a programming system for automation. In *Proceedings 4th International Joint Conference on Artificial Intelligence* (Tbilisi, USSR, Sept. 1975), 758–765.
6. FINKEL, R. Constructing and debugging manipulator programs. Stanford Artificial Intelligence Lab. Memo AIM-284, Stanford, Calif., Aug. 1976.
7. GINI, G., AND GINI, M. Ada: A language for robot programming? *Comput. Ind. 3*, 4 (1982), 253–259.
8. GINI, G., AND GINI, M. Explicit programming languages in industrial robots. *J. Manufacturing Syst. 2*, 1 (1983), 53–60.
9. GINI, M., AND GINI, G. Towards automatic error recovery in robot programs. In *Proceedings 8th International Joint Conference on Artificial Intelligence*, (Karlshrue, West Germany, Aug. 1983), 821–823.
10. GROSSMAN, D. D., AND TAYLOR, R. H. Interactive generation of object models with a manipulator. *IEEE Trans. Syst. Man Cyber. SMC-8*, 9 (1978), 667–679.
11. HASEGAWA, T., AND INOUE, H. Modelling and monitoring a manipulation environment. In *Proceedings 6th International Joint Conference on Artificial Intelligence*, (Tokyo, Aug. 1979), 369–371.
12. KONOLIDGE, K., AND NILSSON, N. Multiagent planning systems. In *Proceedings National Conference on Artificial Intelligence AAAI-80*, (Stanford, Calif., Aug. 1980).
13. LIEBERMAN, L. I., AND WESLEY, M. A. AUTOPASS: An automatic programming system for computer controlled mechanical assembly. *IBM J. Res. Dev. 21*, 4 (1977), 321–333.
14. LOZANO-PEREZ, T. Robot programming. *Proc IEEE 71*, 7 (1983), 821–841.
15. LOZANO-PEREZ, T., MASON, M. T., AND TAYLOR, R. H. Automatic synthesis of fine-motion strategies for robots. *Int. J. Robot. Res. 3*, 1 (1984), 3–24.
16. LUH, J. Y. S. An anatomy of industrial robots and their controls. *IEEE Trans. Autom. Control AC-28*, 2 (1983), 133–153.
17. MUJTABA, M. S., AND GOLDMAN, A. AL *Users' Manual.* Stanford Artificial Intelligence Lab. Memo AIM-323, Stanford, Calif., Jan. 1979.
18. NITZAN, D., AND ROSEN, C. A. Programmable industrial automation. *IEEE Trans. Comput. C-25*, 12 (1976), 1259–1270.
19. PAUL, R. P. WAVE: A model-based language for manipulator control. *Ind. Robot 4*, 1 (1977), 10–17.
20. PAUL, R. P. *Robot Manipulators: Mathematics, Programming and Control.* MIT Press, Boston, Mass., 1981.
21. POPPLESTONE, R. J., ET AL. An interpreter for a language for describing assemblies. *Artif. Intell. 14* (1980), 79–107.
22. RIEGER, C., ROSENBERG, J., AND SAMET, H. Artificial intelligence programming languages for computer-aided manufacturing. *IEEE Trans. Syst. Man Cyber. SMC-9*, 4 (1979), 205–206.
23. ROSEN, C. A., AND NITZAN, D. Use of sensors in programmable automation. *Comput. Mag. 7*, 12 (1977), 12–23.
24. ROSENSCHEIN, J. S. Synchronization of multiagent plans. In *Proceedings National Conference on Artificial Intelligence AAAI-82*, (Pittsburg, Pa., Aug. 1982), 115–119.
25. SALMON, M. SIGLA: The Olivetti Sigma robot programming language. In *Proceedings 8th International Symposium on Industrial Robots*, (Stuttgart, May 1978), 358–363.
26. SANDEWALL, E. Programming in an interactive environment: The LISP experience. *ACM Comput. Surv. 10*, 1 (1978), 35–71.
27. SHIMANO, B. E., GESCHKE, C. C., AND SPALDING, C. H., III. VAL II: A new robot control system for automatic manufacturing. In *Proceedings IEEE International Conference on Robotics*, (Atlanta, Ga., Mar. 1984), 278–292.
28. SMITH, R. G., AND NITZAN, D. A modular programmable assembly station. In *Proceedings 13th International Symposium on Industrial Robots*, (Chicago, Ill., Apr. 1983), 5.53–5.75.
29. TAYLOR, R. H. A synthesis of manipulator control programs from task-level specifications. Artificial Intelligence Lab. Memo AIM-282, Stanford Univ., Stanford, Calif., July 1976.
30. TAYLOR, R. H., SUMMERS, P. D., AND MEYER, J. M. AML: A manufacturing language. *Int. J. Robot. Res. 1*, 3 (1982), 19–41.

31. VERE, S. A. Planning in time: Windows and durations for activities and goals. *IEEE Trans. Pattern Anal. Mach. Intell. PAMI-5*, 3 (1983), 246–267.
32. WILKINS, D. E. Representation in a domain-independent planner. In *Proceedings 8th International Joint Conference on Artificial Intelligence*, (Karlshrue, West Germany, Aug. 1983), 733–740.

Received June 1982; revised June 1984; accepted November 1984