MECHANIKA TEORETYCZNA
I STOSOWANA
Journal of Theoretical
and Applied Mechanics
3, 31, 1993

ROBOT OBJECT-ORIENTED PASCAL LIBRARY: ROOPL

CEZARY ZIELIŃSKI

Institute of Automatic Control
Warsaw University of Technology

The paper presents a manipulator level robot programming language ROOPL implemented as a library of *objects* (*records* and *methods*) in an object oriented programming version of Pascal. ROOPL is located in a classification of robot programming languages. Moreover a brief introduction to object oriented programming methodology is supplied. The robotic system that the language was implemented on is described. A fragment of an exemplary ROOPL program illustrating an execution of simple robotic task is given.

1. Introduction

Initially complex industrial robots were programmed by teach-in methods, but since then robot programming has evolved toward robot specific textual programming languages, Zieliński (1992b). So many robot programming languages (RPL's in short) have been implemented, Zieliński (1989) that classification of both the languages and the methods of their implementation are necessary.

1.1. Classification of robot programming languages

Every programming language operates on specific abstract concepts. An instruction of a language is composed of one or more keywords and zero or more arguments. These arguments express abstract concepts. Computer programming languages (CPL's in short) operate on variables of different types. The values of these variables describe the state of certain abstract notions. The instructions, and so the languages, can be classified according to the abstract notions they refer to.

The main instructions of RPL's are the instructions causing the motion of the manipulator (i.e. motion instructions) and due to that are the reason for motion of the objects in the environment and the robot itself. The abstract notions that

these instructions refer to are: the manipulator joints, the end effector or the objects of the work space. Each of the enumerated notions creates a certain **virtual environment**, Zieliński (1985), (1990) and (1991a), in which the instructions of the language operate. In other words, the elements (considered important) that were selected from the real environment constitute the **virtual environment**. Only some elements of the real environment are the basis for creating abstract notions that compose the virtual environment. The virtual environment is a simplified model of the real environment. On the degree of this simplification and on the abstract notions that were chosen to make up the virtual environment depends the complexity of the control system of a robot. The programmer through each level of the control structure perceives the real environment as a simplified model – he perceives the virtual environment. In the case of RPL's the above mentioned abstract concepts are hierarchically related and so these languages can be classified into levels (Zieliński, 1985; Gini and Gini, 1982).

The languages of the lowest level are called **joint level languages**. The instructions of those languages cause the generation of sequences of signals controlling the drives of the manipulator. So in this case the manipulator joints form the virtual environment. The design of a control system accepting these instructions is quite routine, but to forecast how will the tool behave when all the drives are in motion is not as simple. For the simplification of design the price of programming complexity is paid.

The languages of the next level free the users from this disadvantage. The main concept of the virtual environment of this level is the manipulator's end effector, so these languages are called the **manipulator level languages**. Although it is easy to predict the trajectory of the robot tool when using languages of this level, the programmer still has to be concerned with the description of all the motions of the manipulator instead of simply stating what actions have to be performed to accomplish the task. Examples of languages of this level are: VAL II (User's Guide, 1986) and WAVE (Paul, 1977).

The instructions of **object level languages** operate in virtual environments composed of models of objects existing in the work space. The programmer states only which objects should be transferred, so that the task will be accomplished. The robot control system using its knowledge of the objects and the relations between them will relocate the manipulator in such a way as to complete the job. From this level onward the programmer does not have to busy himself with the motions of the robot arm, but can concentrate on the operations that have to be executed. RAPT (Ambler and Corner, 1984), AL (Mujtaba and Goldman, 1979), TORBOL (Zieliński, 1991b) and SRL (Blume and Jakob, 1986) belong to this level of languages.

On the fourth level instead of specifying all the operations, only a general description of the goal should suffice. In this case the control system has to generate

the plan of actions, and later carry it out. The **task level languages** are the aim of current research. As the prime difference between third and fourth level languages is that expressing tasks in the former we supply the plan of actions and in the later the plan is generated automatically.

RPL can be implemented in three ways

- as a specialised language
- as an enhancement of an existing CPL
- as a robot specific library of procedures coded in a universal CPL.

Implementation of a specialised language is a very laborious task. First the definition of the language (syntax, semantics) has to be elaborated. Usually it turns out that such a language has to possess all the properties of a CPL plus robot specific instructions and data types, what renders it very complex, both to master and even more so to implement. VAL II (User's Guide, 1986), WAVE (Paul, 1977), RAPT (Ambler and Corner, 1984), AL (Mujtaba and Goldman, 1979), TORBOL (Zieliński, 1991b), SRL (Blume and Jakob, 1986) and many other RPL's were implemented in this way.

As a RPL has to have nearly all the constructs attributed to a CPL it seems that the second of the enumerated ways of implementation would be more appropriate. Unfortunately very seldom the definition of a CPL can be enhanced – mainly because the code of the compiler is available only in an executable (non-modifiable) form (the source code is a trade secret), and so this way is usually closed to robotics researchers.

The last method is the cheapest, both in terms of funds necessary for the development of a RPL and the time spent on this development. Only robot specific procedures have to be coded, while all the mechanisms of a CPL are still available to a programmer. Moreover no modification to the compiler or interpreter of the language is necessary. The only drawback is that robot specific instructions are a bit more cryptic (procedures with adequate parameters have to be used instead of explicit robot instructions with appropriate arguments). If library creation is chosen as the means of implementing a RPL, a CPL that will be the foundation, and the programming methodology, still remain to be selected. For instance PASRO (Blume and Jakob, 1985) and ROOPL (Zieliński, 1992c) are submerged in Pascal (Programmer's Guide, 1990), and C (Kernighan and Ritchie, 1980; Turbo C++, Programmer's Guide, 1990) is the basis for RCCL (Heyward and Paul, 1986) and the language of the research oriented controller, Zieliński (1992a).

1.2. Object-oriented and structured approach to programming

Object oriented programming (OOP) methodology evolved from structured

programming. **Structured programming** is a method of describing a programming task in a hierarchy of modules, each describing in increasing detail the task, until the final stage of coding is reached (programming by stepwise refinement). Strict adherence to modules renders GOTO instructions unnecessary, in effect exhibiting a clear program structure. Nevertheless initially structured programming treated data and algorithms operating on this data as two separate entities. Object oriented programming paradigm integrates the two. An **object** is a collection of *data* (variables of adequate type, which should be treated as *fields* of a *record*) and *procedures* and *functions*, which are called **methods**, operating on these variables. Three main properties characterise an **object oriented programming language**

- **Encapsulation** – treating *data* and *code* operating on it as one entity – an *object*;
- **Inheritance** – defining a hierarchy of *objects* in which each *descendant object* acquires all the properties of the *ancestor objects* (access to *data* and *code* of the *ancestors*) and receives some new properties specific to the newly created *object*;
- **Polymorphism** – using the same name for an action that is carried out on different *objects* related by *inheritance*. The action is semantically similar, but it is implemented in a manner appropriate to each of the individual *objects* of the hierarchy.

Some of the CPL's were created as OOP languages (e.g. Loglan, cf Bartol et al., 1982), others which had been originally used only as structured programming tools were enhanced by adding OOP mechanisms (e.g. Pascal, Programmer's Guide, 1980; C++, Programmer's Guide, 1990).

OOP methodology assumes that certain abstract *objects* will be defined by the programmer. These *objects* have their properties (*data*) and exhibit behaviours (*methods*). The program is written in terms of *objects* behaving in such a way as to change their properties, i.e. applying *methods* to change *data*. To make this more clear an example follows. Let a screw be the *object*. One of its many characteristics is its location in space (*data*). The screw can move in space (movement is its behaviour). The programmer commands the screw to change its position, and so applies its position changing *method* to its *data*. As the result of applying this *method* the screw will be transported to some other location.

At this point the misunderstanding which can arise from the traditional use of the term "*object*" in "*object level robot programming languages*" and in "*object oriented programming languages*" (this time CPL's) has to be clarified. In the case of RPL's the notion of an "*object*" pertains to the real objects that are located in the robot's environment or to abstract models of these objects represented in

a RPL. In the case of the CPL's the term "*object*" represents an abstract notion, which encapsulates *data* and *code*, and possesses the properties of *inheritance* and *polymorphism*.

This paper describes the application of OOP methodology to the creation of a manipulator level RPL (*object* library to be strict). For this purpose a version of Pascal language possessing OOP enhancements (Turbo Pascal, Programmer's Guide, 1980) was used.

2. ROOPL

2.1. Implementation

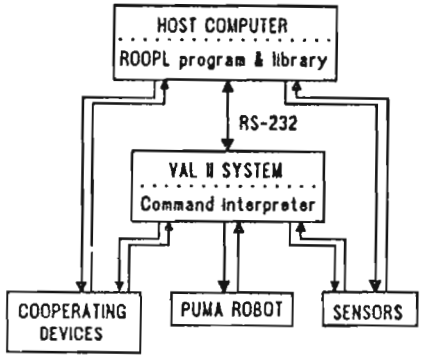

Fig. 1. Structure of the system executing ROOPL programs

ROOPL (Robot Object Oriented Pascal Library) is a library of *objects* and *methods*, coded in an OOP version of Pascal (Programmer's Guide, 1980), that can be used in programs generating, modifying and executing robot arm trajectories.

The structure of the system executing ROOPL programs is shown in Fig.1. A ROOPL program is executed on an IBM PC class host computer. It generates robot motion commands that are being interpreted by a VAL II (User's Guide, 1986) program running on a robot control system computer. Both computers are connected by a RS-232 serial interface. The VAL II control system causes the motion of a PUMA-560 robot arm. The sensors can be either directly connected to the host computer or to the control computer. In the later case, data obtained from sensors is transmitted through the RS-232 interface to the host computer for processing. The RS-232 is also used for transmitting to the host computer the information about the current state of the arm (e.g. about motion termination or

the arm pose). ROOPL programs can also control cooperating devices connected either to the host computer directly or indirectly through the control computer.

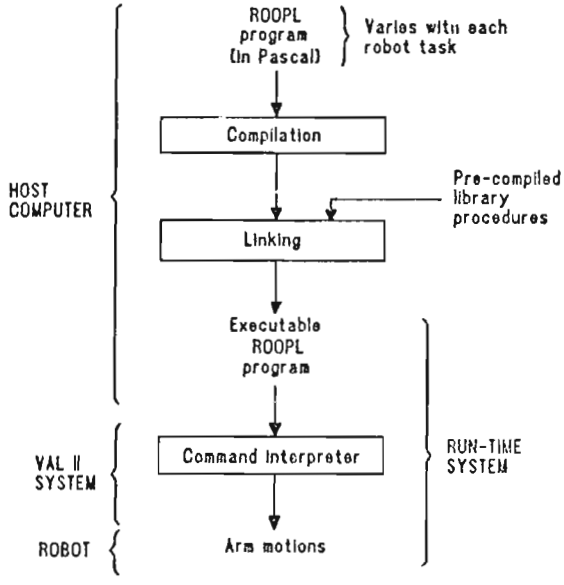

Fig. 2. Method of processing a ROOPL program

The method of processing and executing ROOPL programs is presented in Fig.2. The source program is written using any text editor. Next the source program is compiled by a Pascal compiler. The resulting object code files and the ROOPL library module are linked, and so the executable code is obtained. This code is run on the host computer. Simultaneously a program called command interpreter, which was written in VAL II, is run on the control computer. After an automatic synchronisation phase through the RS-232 interface, the robot task initially coded in ROOPL is executed. The command interpreter constantly waits for motion commands and status sensor requests from the host computer. In response to these commands it initiates the execution of motions and sends back the requested data and information about motion termination.

2.2. General information about the ROOPL library

To use ROOPL, a Pascal program invoking library *objects* and their *methods* has to be written. At its beginning it should contain the following clause:

```
uses roopl;
```

Homogeneous transform matrix type representing a dextrorotatory set of orthogonal unit axis (coordinate frame) - `matrix`, and a pointer type - `MatrixPtr` to such a frame, are defined as supplementary data types. As a homogeneous transform matrix can also represent transformations (translation and rotation) no other robot specific data type need to be introduced.

Two types of *objects* are defined by the library: `frame` and `segment`. The first represents a homogeneous transform representation of a coordinate frame. The later is a *descendant* type of the former and describes the approach path `segment` to the *ancestor* `frame`. Their definitions are given in Fig.3 and Fig.4.

```
frame = object
  location:    MatrixPtr;
  constructor  Create(px,py,pz,fi,theta,psi: real);      {Create
                      the frame using Cartesian-Euler description}
  destructor   Destroy; virtual;          {Release memory}
  procedure    Copy(F: frame);            {Copy F.location into
                                          self.location}
  procedure    Invert;                    {Invert self.location}
  procedure    LeftMultiply(F: frame);    {left multiply self.
                                          location by F.location}
  procedure    RightMultiply(F: frame);   {right multiply self.
                                          location by F.location}
  function     Equal(F: frame):           {Compare self.location}
               boolean;                   {and F.location}
  procedure    WriteLocation;             {Write out self.location}
end; frame
```

Fig. 3.

The *methods* defined in the `frame` *object* (`Copy`, `Invert`, `LeftMultiply`, `RightMultiply`, `Equal`) perform the obvious homogeneous matrix operations. The first argument of the operations is the `location` *field* of the *object* and the second (where present) is defined by *method*'s actual parameter.

```
segment = object (frame)
  motion:          MotionType;              {type of motion to
                                          be executed along the segment}
  Sensor_LSB:      byte;                    {VAL input number that
                                          is to be treated as a LSB}
  Sensor_reading:  integer;            {value obtained from the sensor}
  Error:           byte;                    {error code (e.g. errors
                                          may occur during transmission}
  constructor      Create(px,py,pz,fi,      {Create segment using}
                   theta,psi: real;         {the listed arguments}
                   SegMotion: MotionType;
                   SensLSB: byte);
  destructor       Destroy; virtual;        {Release memory}
  procedure        AttractPumaTool          {Move robot; A6T -}
                   (A6T: frame);            {Flange-Tool transform}
  function         SensorData: byte;        {Read sensor data byte}
  function         ErrorOccurred: Boolean;  {Get error}
  procedure        WriteError;              {Write out error}
  function         SegmentType:             {Get segment type}
                   MotionType;
  procedure        Homogeneous_to_Euler     {Transform homogeneous}
                   (var px,py,pz,fi,        {to Cartesian-Euler}
                   theta,psi: real);        {representation}
  procedure        Copy(S: segment);        {Copy S into self}
  procedure        SetLocation(px,py,pz,    {Set location field}
                   fi,theta,psi: real);     {to Cartesian-Euler}
                                            {representation}
  procedure        InitRobot(A6T: frame);   {Initialize the robot}
  procedure        QuitRobot(A6T: frame);   {Deactivate the robot}
  function         GetSensorData: byte;     {Get sensor reading}
  function         SensorNumber: byte;      {Get sensor number}
  procedure        SetSegmentType(m:        {Set segment type}
                   MotionType);
  procedure        SetSensorNumber(sn:      {Set sensor LSB number}
                   byte);
end; segment
```

Fig. 4.

There is only one, but general, *method* of moving the robot. The `AttractPumaTool` *method* applied to a `segment` *object* causes the robot to move towards (be attracted by) the coordinate frame being one of the *fields* of the *object.*

The type of motion depends on the contents of the `motion` *field*. Eight different kinds of motions can be performed – defined by enumerated type `MotionType`.

- `PTP_JointInterpolated_NoSensors` causes joint interpolated motion without using sensors; confirmation is sent when the motion terminates
- `CP_JointInterpolated_NoSensors` causes joint interpolated motion without using sensors; confirmation is sent when the motion is initiated
- `PTP_JointInterpolated_WithSensors` causes joint interpolated motion with sensor feedback; confirmation is sent when the motion terminates
- `CP_JointInterpolated_WithSensors` causes joint interpolated motion with sensor feedback; confirmation is sent when the motion is initiated
- `PTP_Cartesian_NoSensors` causes Cartesian interpolated motion without using sensors; confirmation is sent when the motion terminates
- `CP_Cartesian_NoSensors` causes Cartesian interpolated motion without using sensors; confirmation is sent when the motion is initiated
- `PTP_Cartesian_WithSensors` causes Cartesian interpolated motion with sensor feedback; confirmation is sent when the motion terminates
- `CP_Cartesian_WithSensors` causes Cartesian interpolated motion with sensor feedback; confirmation is sent when the motion is initiated

In the case of PTP (point to point) motion its termination is signaled, while in the case of CP (continuous path) motion its initiation is signaled by the VAL command interpreter. Either straight line in the joint angle coordinates or in Cartesian-Euler space can be used while approaching the goal frame. If sensors are used during the motion, the confirmation byte carries the information about the obtained sensor reading. Otherwise an asterisk is sent as confirmation token. Sensor reading is obtained from the 8 consecutive VAL inputs, starting at Sensor_LSB (inclusive).

Any *method* that can finish its execution without performing its task causes the `Error` *field* of a `segment` to change its value to non zero. This *field* should be checked whenever executing an action that can result in an error (e.g. transmission error).

Other *methods* defined in `segment` manipulate the *data fields* of this *object* (e.g. read them). It is easier for the programmer to describe frames as three Cartesian coordinates of the origin and three Euler angles of orientation, so adequate *methods* for transforming the internal format into this representation and vice versa are supplied (`Homogeneous_to_Euler`, `SetLocation`).

As can be seen from the above definitions nearly all the actions are performed by executing appropriate *methods* on the two supplied *object* types (`frame` and `segment`). Only gripper closing and opening is done by procedures.

```
procedure Grasp(var Err:  byte);
procedure Release(var Err:  byte);
```

The robot program consists of sequences of *methods* executing actions on *objects*. Besides these the programmer is free to use any Pascal statements.

2.3. Example

A fragment of a program executing a path that is generated on-line will be presented as an illustration of program coded in ROOPL, Fig.5. In the case of sensor guided motions each new location depends on the value obtained through a set of binary sensors connected to the VAL system. Eight kinds of motion can be used by selecting one from each of the three following pairs: (PTP, CP), (joint interpolated, Cartesian interpolated), (with sensors, without sensors). The *object* Seg is the definition of the current path segment. It is modified for each next motion step. As a result of this program the operator can lead the robot by exerting forces on the sensor to a goal location that has non-decreasing coordinates in relation to the current location (this assumption was made to keep the example short). The sensor can be mounted near the tool tip or be placed in any other area.

3. Conclusions

The paper presents an object oriented programming approach to the definition of a manipulator level robot programming language. The language was defined as a library of *objects* (*records*, *methods*), specific to robot programming, and submerged in a universal computer programming language: OOP Pascal. The library takes into account both robot motions and sensor data processing. The language was implemented on a two computer system controlling PUMA-560 robot. The performance of the system was satisfactory in the case of predefined motions that do not have to be frequently modified by actions dictated by sensors. In the case of trajectories that are frequently modified or generated according to data obtained through sensors, due to low transmission rate of the RS-232 serial interface, a jerky motion can result. This drawback can be easily solved by employing a parallel interface and increasing the transmission rate.

```
{Execute 15 motion steps}
for i:=1 to 15 do
begin

 {Create the next location taking into account the sensor data}
 case Seg.SensorData of
 {Sensor supplies a number:  0 - 7}
 0:  begin  {do nothing}                                          end;
 1:  begin  x := x + xstep;                                       end;
 2:  begin                  y := y + ystep;                       end;
 3:  begin  x := x + xstep; y := y + ystep;                       end;
 4:  begin                                  z := z + zstep;       end;
 5:  begin  x := x + xstep;                 z := z + zstep;       end;
 6:  begin                  y := y + ystep; z := z + zstep;       end;
 7:  begin  x := x + xstep; y := y + ystep; z := z + zstep;       end;
 end {case Seg.SensorData}
 Seg.SetLocation(x,y,z,PI/2,PI/2,0);

 {Execute the current step}
 Seg.AttractPumaTool( A6T );

 {Check if an error occurred}
 if Seg.ErrorOccurred
  then
   begin
    Seg.WriteError;
    Seg.Destroy;
    halt;
   end;

end; {for}
```

Fig. 5.

Acknowledgment

The implementation of ROOPL at Loughborough University of Technology was supported by the Science and Engineering Research Council Grant no. GR/G64091. Some parts of the presented research were also supported by statutory research funds of the Warsaw University of Technology. The author gratefully acknowledges both sources of funding.

References

1. AMBLER A.P., CORNER D.F., 1984, *RAPT1 User's Manual,* Department of Artificial Intelligence, University of Edinburgh
2. BARTOL W.M. ET. AL., 1982, *Report on the LOGLAN 82 Programming Language,* Institute of Informatics, University of Warsaw
3. BLUME C., JAKOB W., 1985, *PASRO: Pascal for Robots,* Springer-Verlag
4. BLUME C., JAKOB W., 1986, *Programming Languages for Industrial Robots,* Springer-Verlag
5. GINI G., GINI M., 1982, *ADA: A Language for Robot Programming?",* Computers In Industry, **3**, 4
6. HEYWARD V., PAUL R. P., 1986, *Robot Manipulator Control Under Unix RCCL: A Robot Control "C" Library,* International Journal of Robotics Research, **5**, 4
7. KERNIGHAN B.W., RITCHIE D.M., 1980, *The C Programming Language,* Prentice-Hall
8. MUJTABA S., GOLDMAN R., 1979, *AL Users' Manual,* Stanford Artificial Intelligence Laboratory
9. PAUL R., 1977, *WAVE - A Model Based Language for Manipulator Control,* The Industrial Robot
10. ZIELIŃSKI C., 1985, *Hierarchy of Robot Programing Languages and the Semantics of Some of their Instructions,* Archiwum Automatyki i Telemechaniki, 3-4, (in Polish)
11. ZIELIŃSKI C., 1989, *Overview of the Features of Existing Robot Programming Languages,* Archiwum Automatyki i Telemechaniki, 3, (in Polish)
12. ZIELIŃSKI C., 1990, *Description of Robot Programming Language Instruction Semantics,* Archiwum Automatyki i Telemechaniki, 1-2, (in Polish)
13. ZIELIŃSKI C., 1991a, *Description of Semantics of Robot Programming Languages,* Mechatronics, **1**, 2, Pergamon Press
14. ZIELIŃSKI C., 1991b, *TORBOL: An Object Level Robot Programming Language,* Mechatronics, **1**, 4, Pergamon Press
15. ZIELIŃSKI C., 1992a, *Flexible Controller for Robots Equipped with Sensors,* Proceedings of the 9-th CISM-IFToMM Symposium on Theory and Practice of Robots and Manipulators, Ro.Man.Sy'92, Udine, Italy
16. ZIELIŃSKI C., 1992b, *Object Level Robot Programming Languages,* in Robotics Research and Applications, ed. Morecki A., Muszyński W. and Tchoń K., Warsaw
17. ZIELIŃSKI C., 1992c, *Programming Languages for Flexible Automation,* SERC Grant Report (GR/G64091), Loughborough University of Technology
18. *Turbo C++: Programmer's Guide, Ver.2.0,* Borland International Incorporated, 1990
19. *Turbo Pascal: Programmer's Guide, Ver.6.0,* Borland International Incorporated, 1990
20. *User's Guide to VAL II: Programming Manual, Ver.2.0,* Unimation Incorporated, A Westinghouse Company, August 1986

ROOPL – język programowania robotów zanurzony w obiektowej wersji Pascala

Streszczenie

W artykule przedstawiono język programowania robotów zorientowany na przemieszczanie manipulatora, zaimplementowany jako biblioteka *obiektów* (*rekordów* i *procedur*) w obiektowej wersji języka Pascal. Język ten umiejscowiono w klasyfikacji języków programowania robotów. Ponadto zamieszczono krótki opis pojęć związanych z programowaniem strukturalnym i obiektowym. Opisano również system składający się z komputera klasy IBM PC, robota PUMA oraz jego sterownika – na którym zaimplementowano wyżej wzmiankowany język ROOPL. W celu ilustracji sposobu programowania w języku ROOPL, zamieszczono fragment programu realizującego proste zadanie wykonywane przez robota wyposażonego w czujniki.

Manuscript received March 5, 1993; accepted for print March 22, 1993