Computer Science 773

Robotics and Real-time Control

ROBOT PROGRAMMING LANGUAGES

The software which drives robots is most commonly written in an ordinary programming language, with C or C++ perhaps the most common. For programming the robots themselves, though, special languages have met with some acceptance, because the operations required – particularly the geometry – are sufficiently specialised to make it worth while. You could, of course, do it in C with a specialised function library, but the result wouldn't be particularly readable.

However it's done, you want lots of support for data structures and appropriate operations. I'll discuss some special robot languages, because the discussion brings out the different levels at which programming can be described and (with a lot more difficulty) performed, but the same considerations apply to doing it with conventional languages.

LEVELS OF PROGRAMMING.

Just as you can give instructions to a computer on many levels, using assembly languages, procedural languages, declarative languages, and so on, so you can express robot instructions in different ways. Here's a list of five levels of increasing sophistication and complexity.

Notice that the idea of "level" here is concerned with the level of description of the process being performed. It isn't the same as the more syntactic notion of level used in classifying conventional programming languages. There's a sort of correspondence, but that's accidental so far as I know.

Actuator level	Instructions are given in terms of the motions of the actuators. This is the equivalent of machine language; everything has to be converted into instructions at this level eventually, but nobody uses it directly.	
Joint level	Instructions are given in terms of joint coordinates. In effect, this is what you get after solving the inverse kinematic problem.	

Manipulator level	Instructions are given in terms of ordinary spatial coordinates. This is the lowest level at which you'd want to write a programme. Compare it with an assembly language.	`move to ( a, b, c );`
		`close gripper;`
		`move to ( d, e, f );`
		`open gripper;`

Task level	Instructions are given in terms of the job to be done. This is beginning to get fairly tricky to implement. Analogous to a procedural language.	`move base to jig;`
		`drill hole in centre;`
		`stand upright on base;`
		`insert screw through`
		`  hole in base into`
		`  threaded hole in`
		`  upright;`

Object level	No instructions – you describe the finished product. The system has to work out the instructions. Almost impossible, but not quite. This is a declarative language, comparable with Prolog.	`screw passes through hole in base into threaded hole in upright`

SOME EXAMPLES.

Here are some examples of small programmes (or fragments thereof) written in a few robot programming languages. They are selected mainly to illustrate different approaches to robot programming, but the selection was biased by the ready availability of examples in certain languages. Nevertheless, I think this set is fairly representative, except for RSS.

WAVE : 1970 – 1975.

R. Paul : "WAVE : a model based language for manipulator control", *The Industrial Robot*, 10 (March 1977).

WAVE's main distinction is its claim to be the first robot programming language. As well as instructions which move the robot, it provides for force control and input from a vision system.

WAVE is a manipulator-level language, with provision for a gripper and – perhaps oddly, in view of its rather low level – force control. Syntactically, it looks rather like a conventional computer assembly language. Think of it as an assembly language for a virtual machine with somewhat unusual registers.

EXAMPLE : A portion of a programme to pick up parts from a stack. STACK and AWAY are the positions of the stack base, and where to put the block; they are set in the first two (incomplete) instructions of the programme. TOP is also predefined, but not shown in the programme; it's the displacement of the top of the stack from its base.

TRANS	defines a spatial position (on first use with a variable) or transformation (if used again with the same variable). TRANS STACK will be followed by six numbers giving a displacement and orientation.
RESTORE	resets the "current transformation" to the value in its argument, which is a thing called a SAVE cell. A SAVE cell holds a transformation, initially a null transformation, but resettable using a SAVE instruction. (I think.) The "current transformation" is a displacement (compare an index register) from a base position, so you can easily use local coordinates for an object in different places.
MOVE	moves the robot's hand to the position identified by its argument as transformed by the current transformation. The intention is to make it easy to perform the same sequence of actions in different places without cluttering the code with lots of necessary but bulky transformations.
CLOSE	closes the gripper to the distance given as the argument.

SKIPE	skips one instruction if an error is signalled with error number matching the argument. Presumably CLOSE fails if it succeeds (if it does close, it signals an error), and only succeeds if it fails – that is, if the gripper can't close to the distance specified.
JUMP	does.
OPEN	opens the gripper to the distance given as the argument.
CHANGE	moves the hand. I can't work out from the text just what the different arguments are. (I think it changes the current transformation.)
SAVE	resets the value of the SAVE cell named. It is cleverer than it looks : it actually resets the transformation rather than the absolute position.

Here is the programme :

```
    TRANS STACK ...      ; Top of the stack of blocks
    TRANS AWAY ...       ; Where to go with block

    RESTORE TOP          ; Move to STACK modified by TOP
    MOVE STACK
TEST:
    CLOSE 1              ; Pick up block
    SKIPE 2              ; Error 2, no block there
    JUMP OK              ; Error did not occur go to OK
    OPEN 5               ; Error did occur, open up again
    CHANGE Z,-1,NIL,0,0  ; Move down one inch
    SAVE TOP             ; Change TOP to get us here
    JUMP TEST            ; and try again
OK:
    MOVE AWAY            ; All is well here
```

EXAMPLE : Compliance.

WAVE will also handle compliance and force control : it has instructions FREE, SPIN, WOBBLE, and FORCE which, respectively, identify directions and axes of free translation and rotation, add a dither to the hand's motion so that it can find a stable position of some sort, and prescribe a force and torque which the hand should exert in the directions of compliance. The example sets compliance and force conditions which must be observed while executing the CHANGE instruction.

FREE	defines the number of degrees of translational freedom and the directions of freedom.
FORCE	defines a force and torque which must be exerted.

```
FREE 1,Z              ; Comply in the z direction
FORCE FV,NIL          ; and exert 50 oz. in the z
                          direction
CHANGE X,2,NIL,0,0    ; while moving 2 inches in x
```

RSS : Robot Servo System : ≈ 1979.

C.C. Geschke : "A system for programming and controlling sensor-based manipulators", *IEEE Trans.Pat.Anal.Mach.Int.* **5**, 393 (1983).

RSS is interesting in that its instructions aren't simply of the "go to (x, y, z)" pattern, but rely on *servo processes* which try to establish relationships between the positions of things in the system, and to maintain the relationships once established. It also provides for integration with sensory systems, particularly vision.

It has elements of manipulator level methods – much of the detail is expressed in quite low-level terms, though extensive use of defined values makes it more readable than one might expect – but the notion of identifying constraints which the system is then expected to maintain approaches the object level. Notice too that this implies a multiprocessing model, as many essentially independent servo processes have to be maintained simultaneously.

The vision system is separate; it is assumed that it maintains something equivalent to a world view, which can be interrogated when information is required. In this example, the vision system knows how to identify bolts and holes, can be instructed to find them, will return an indication (called flag) when the search is successful or when anything goes wrong, and can track an object as it moves. It also returns the coordinates of the objects which it finds.

Anything in the programme beginning with R$ or r$ is a reserved symbol to do with the position or orientation of the robot hand. The symbols, and some others associated with the workpiece, are defined in the diagrams below.

EXAMPLE : A programme to put a bolt into a hole.

SERVO	turns ON or OFF all the servo functions.
FORCE and TORQUE	are servo functions which maintain a force or torque at the robot wrist.
VISION	declares that its arguments are functions with values provided by the vision system. It is the vision system's job to recognise these objects (the paper doesn't say how, but gives a reference) and return the coordinates when required.
DEFINE	defines a function, the value of which is maintained by the system as the robot's configuration changes.
LOCATE	is an instruction to the vision system to find the argument and set the value of a variable to its coordinates
ORIENT FIXED	defines a servo process which holds the orientation of the two axes defined at the values defined, thereby fully defining the orientation. ORIENT by itself defines only one axis, allowing freedom in other directions.
POSITION POINT	defines a position servo process.
TRAP	deals with error signals.
TRACK	instructs the vision system to keep track of the argument and continually adjust the value to reflect the current position.
POSITION LINE	defines a servo process which constrains its argument to move to and along a defined line; this type of process can be used to control compliant motion.
FORCE	is used for force control.

```
; Bolt insertion routine
; Clifford C. Geschke 9-12-78
    servo off
    force ZERO
    torque ZERO
; Define some functions
    vision BOLT, HOLE
    define HOLEAXIS = [0,0,1]
    define BOLTAXIS = R$FINGER
    define BOLTPOS = R$GRIP + 2 * BOLTAXIS
    define BOLTGOAL = HOLE + 2 * HOLEAXIS
    define ERROR = |(BOLTPOS – HOLE) # HOLEAXIS|
    define f = 25
; Tell vision processor to locate HOLE
    locate HOLE
    wait until flag found HOLE
    print HOLE found.
; Move to above estimated hole location
    orient fixed BOLTAXIS;R$THUMB = – HOLEAXIS;[-1,0,0]
    position point BOLTPOS = BOLTGOAL
    servo on
    wait until |BOLTPOS – BOLTGOAL ;lss .5;
; Locate and track BOLT
    trap 2 to lostit on flag lost BOLT
    locate BOLT
    wait until flag found BOLT
    print BOLT found
    track BOLT
    define BOLTPOS = BOLT
    position line BOLTPOS = HOLE;HOLEAXIS
; Insert BOLT into HOLE
fwait:  force ZERO
    wait until error ;lss .1
    trap 1 to fwait on error ;gtr .1
    force f*BOLTAXIS
    wait until R$FORCE·BOLTAXIS ;lss –.8*f
    print All done!
    stop
;
; Error routine if BOLT is lost
lostit: servo off
    print Lost BOLT
    stop
```

AUTOPASS : 1977.

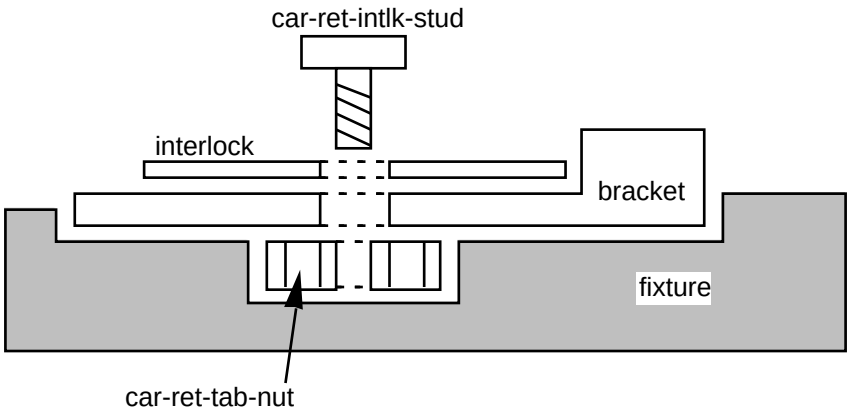
L.I. Lieberman, M.A. Wesley : "AUTOPASS : an automatic programming system for computer controlled mechanical assembly", *IBM J.Res.Dev* **21**, 380 (1977).

A high level language for describing assembly operations in terms of the objects for assembly. An Autopass programme gives a sequence of instructions for assembly, with descriptions of how the parts are to be fitted together. The compiler is required to work out the correct sequence of instructions for the robot.

Autopass is clearly a task-level language.

EXAMPLE : assembling a support bracket.

The programme is fairly self-explanatory – though obviously it needs a lot of declarations and definitions. The support bracket itself is never clearly described in the reference, but this is my reconstruction :



That picture carries no guarantee, but it does make sense of the programme:

1. OPERATE *nutfeeder* WITH *car-ret-tab-nut* AT *fixture.nest*
2. PLACE *bracket* IN *fixture* SUCH THAT *bracket.bottom* CONTACTS *car-ret-tab-nut.top* AND *bracket.hole* IS ALIGNED WITH *fixture.nest*
3. PLACE *interlock* ON *bracket* SUCH THAT *interlock.hole* IS ALIGNED WITH *bracket.hole* AND *interlock.base* CONTACTS *bracket.top*
4. DRIVE IN *car-ret-intlk-stud* INTO *car-ret-tab-nut* AT *interlock.hole* SUCH THAT TORQUE IS EQ 12.0 IN-LBS USING *air-driver* ATTACHING *bracket* AND *interlock*
5. NAME *bracket interlock car-ret-intlk-stud car-ret-tab-nut* ASSEMBLY *support-bracket*

RAPT : 1978.

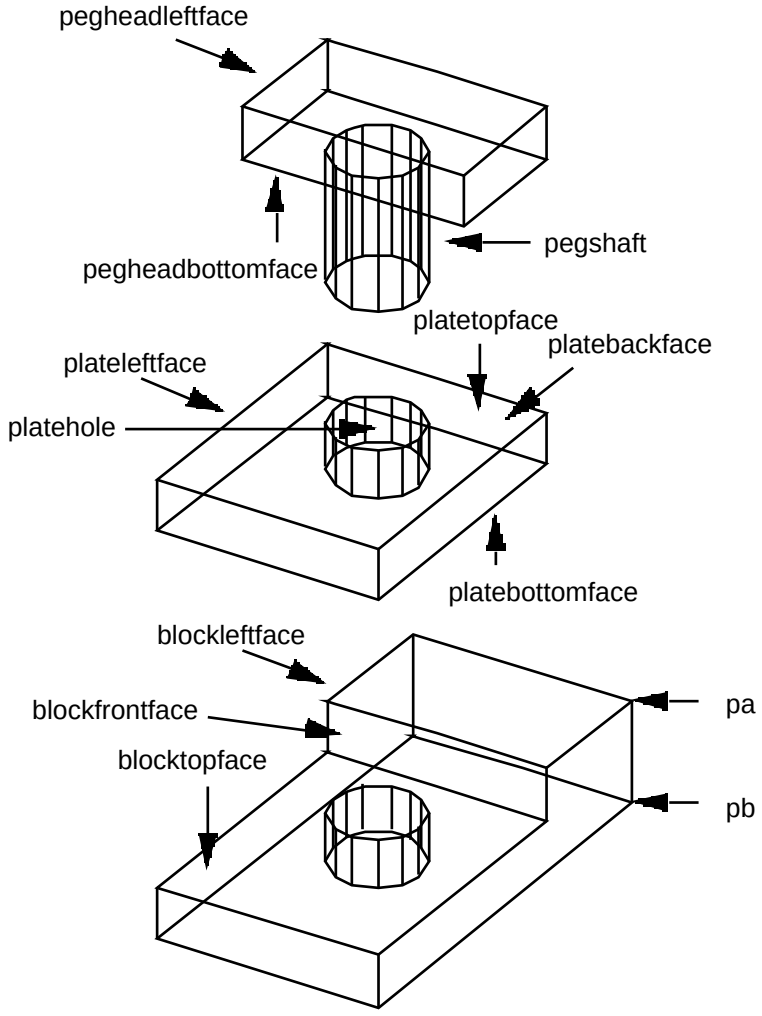
A.P. Ambler, S.A. Cameron, D.F. Corner : "Augmenting the RAPT robot language", *DAI Research Paper* **#330**, Department of Artificial Intelligence, Edinburgh University, 1986.

RAPT was originally developed in the style of APT (I have no idea why; someone suggested to me that it was a bid for respectability), and was intended as a very high-level language for describing assembly tasks. It gives the same sort of information as Autopass, but if anything even less about how to do the assembly. In the example, the only actual instructions are two MOVEs, and the rest is all geometry. It is worth remarking, though, that the task which the programme accomplishes is trivial.

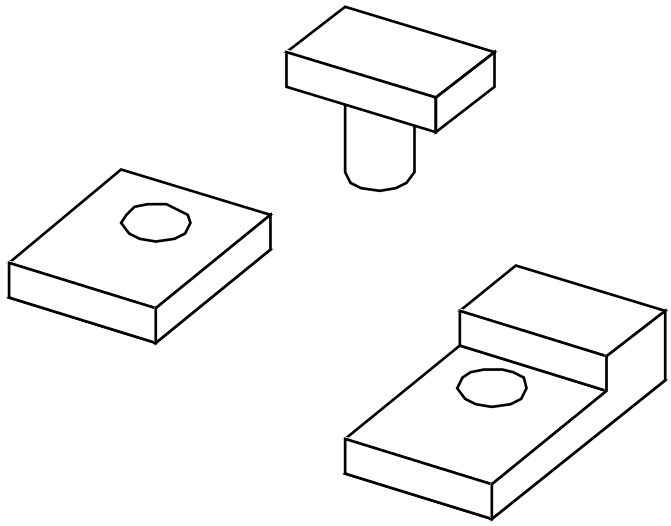
RAPT is unambiguously aimed at the object level.

EXAMPLE : a simple assembly task.

The plate is to be attached to the block using the peg in the obvious way. Note that I have drawn the pointers on the diagram (which I have also redrawn, but reasonably accurately), as they were missing from my copy of the report. I think they're right.



This is what the separate parts look like :



- and here is the RAPT programme :

```
tied/block, world     [ this specifies that the block is fixed in the world ]
move/plate   [  this specifies that the plate should be moved  ]
move/plate, parallelto dir1, 20
                [ followed by a move of 20 units in the direction dir1 ]
                [ The next few lines specify where the plate must be now; RAPT
                will use this to compute the two moves described above ]
coplanar/ plateleftface, blockleftface
against/ platebottomface, blocktopface
against/ platebackface, blockfrontface
                [ Now we will specify a similar sequence for the peg ]
move/ peg
move/ peg, parallelto dir1, 40
against/ pegheadbottomface, platetopface
coaxial/ pegshaft, platehole
parallel/ pegheadleftface, plateleftface
                [ now define dir1 as the vector between two points ]
dir1 = line/pa,pb
```

WHAT'S HAPPENING NOW.

So far as I can make out, nobody is pushing very hard at the moment to advance these rather ambitious languages. There is a feeling that they've gone about as far as they can go. That's because the sort of structural information that you give in a RAPT programme is much the same sort of thing as you can get from CAD systems, and to handle anything but the simplest task by writing painstaking instructions for every step is very hard work.

And they don't help at all when you want to write programmes for advanced robots which have to deal with arbitrarily complicated contingencies which might occur in uncontrolled environments. They must necessarily depend heavily on perception of the world in working out how to achieve goals, and the specialised languages are not obviously well suited to that sort of activity.

The consequence is that teaching has largely taken over for simple repetitive tasks, but tasks which are too complicated for teaching are also likely to be too complicated for the special languages. In fact, programming a robot is not really very like programming a computer; the environment is far more important. There's no equivalent in conventional computing to knocking a workpiece off the bench.

Alan Creak,
April, 1998.