High-Level Robot Programming in a PC-Based Control Environment

Matteo Malosio, Matteo Finardi, Simone Negri, Lorenzo Molinari Tosatti and Francesco Jatta
Istituto di Tecnologie Industriali e Automazione
Consiglio Nazionale delle Ricerche, Milano
{m.malosio, m.finardi, s.negri, l.molinari, f.jatta}@itia.cnr.it

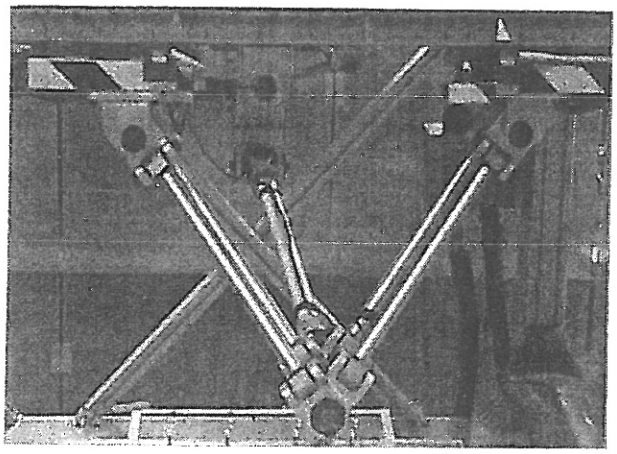

Fig. 1. The 4 d.o.f. PKM used as a testbed for the control

Abstract – **In the last few years, the wide availability of computing power and operating systems with real-time capabilities offered the chance to develop low-cost PC-based solutions for motion control; this made possible to overcome limits in commercial robot controllers for their use in a research context. Moreover, modern scripting languages are becoming more and more powerful. High-level programming permits to easy write code in a smart way and does not require particular skills. In this paper we present the design of a modular architecture for PC-Based robot control under the QNX Real Time operating system. The proposed architecture allowed to use the Python programming language as an high-level object oriented scripting tool to program robot motions and to monitor, even remotely, the system. The designed architecture has been employed for the motion control of a 4 d.o.f. reconfigurable parallel robot, and the advantages deriving from the use of a high-level object-oriented robot programming language are shown through the programming of a demonstrative assembling cell of a manufacturing plant.**

I. INTRODUCTION

Nowadays industrial companies have to face frequent and unpredictable market changes. So, to remain competitive, companies must possess a new type of production system and intelligent machines able to react to such changes.

A limit for a broader employment of robots in this context is represented by the use of proprietary hardware, closed software architectures and communication protocols implemented by the robot manufacturers. In particular commercial robot controllers are programmable via their own procedural languages that run on proprietary hardwares. This feature limits the possibility to extend the robot capabilities, and the implementation of advanced sensing capabilities, as force control or vision systems, is not a trivial task. In the last years the wide availability of low cost computing power, interface cards for data acquisition and operating systems with real-time capabilities made, in the last decade, PC-based solutions for robot motion control feasible [1]–[6].

Beside universities and research laboratories, even the industrial world of numerical control manufacturers is looking with interest to PC-based solutions as demonstrated by their membership in research consortia aimed at the purpose of proposing standards for open architecture controllers: OSACA[1], OMAC[2], OSEC[3] and OROCOS[4], [7].

Despite the efforts that have been put so far, no widely recognized standards arose, neither in terms of reference architectures nor in terms of software tools (operating systems and programming languages).

Moreover, scripting languages are growing in popularity inside the programmers' community and their characteristics makes programming more and more simple; their diffusion is increasing in the computer world, but their adoption in the industrial world is at the beginning. Nowadays robot programming is still related to elementary structures and limited by low level structures and functions. Despite programming easiness and high-level characteristics of modern scripting languages, they are not yet used to program robot, excluding a few examples [8]–[10]. In our research we evaluated the programming features of different scripting languages; finally our choice has been in favour of Python [11]. In this work we exploit the benefits deriving from object-oriented design, very high-level structures and rich libraries, some of Python key features [12], [13].

In this paper we analyze how the Python programming environment has been developed and interfaced with a PC-based control system, whose core is a QNX real-time platform programmed through C++ code.

The designed solution has been applied for the motion control of a novel 4 d.o.f. reconfigurable parallel kinematic machine, thus exploiting the benefits deriving from the defined modular control architecture [14].

The work is organized as follows: Section *II* illustrates a general high-level overview of the control system, in Section *III* the correspondent software implementation is detailed with reference to the QNX4 operating system, in Section *IV* the use of Python in the programming and monitoring functionalities of the system is explained and a demonstrative application of the Python-based programming system is reported in Section *VII*.

Finally, conclusions and future works are drawn in Section *VIII*.

[1] Open System Architecture for Controls within Automation systems
[2] Open Modular Architecture Controller
[3] Open System Environment for Controllers
[4] Open RObot COntrol Software

II. CONTROL SYSTEM COMPONENTS

With reference to Fig. 2 the whole control system can be subdivided in the following basic components.

The *Interpreter* takes care of decoding a high-level motion command specified by the user into a well defined data structure containing the detailed specification of the motion command, i.e.:

- geometrical data:
  - interpolation path that connects the current location to the target one (linear, circular...);
  - target coordinates (position and orientation of the tool center point (TCP)) and corresponding derivatives;
- frame data: specifies the frames under which the geometrical data are referred to;
- motion profile data:
  - motion of the TCP specified in terms of its corresponding acceleration profile (sine, bang-bang, ...);
  - velocity values at the motion end-points;
  - peak velocity of the TCP motion profile.

All the steps involved at this stage have been implemented through the Python programming language (Section IV).

A very first level of safety check is performed at this level to verify the physical consistency of the user data like joint position and velocity limits.

The *Trajectory Generator* acts through three different levels:

- first a Cartesian path is generated according to the geometrical data specified in the motion data structure;
- the Cartesian path is then sampled according to the motion data profile;
- finally the sampled trajectory is transformed via kinematic inversion to actuators coordinates setpoints.

Servo-Controller. Its aim is to keep the actual actuators coordinates as close as possible to the corresponding setpoints by means of a proper control algorithm.

The *Interface-driver* is the software interface component to the interface card that allows bi-directional data communication between the controller and the robot. Its functionality is twofold:

- it provides time synchronization to the whole control system;
- allows sensors signal readings (encoders, analog and digital inputs) and the writings of command current setpoints.

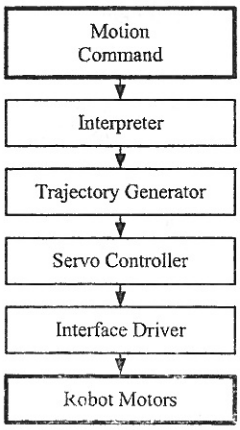

Fig. 2. Control system components

III. SOFTWARE IMPLEMENTATION OF THE CONTROL

A. The QNX4 operating system

The core components of the high-level scheme described in Section II have been implemented under QNX4. QNX4 is a commercial real-time operating system manufactured by QSSL, characterized by a micro-kernel architecture [15]. The kernel itself is devoted only to a restricted number of services to the other processes: it provides a message passing mechanism and performs scheduling activities. The whole operating system is constituted of modules built on top of the kernel communicating with each other by means of messages. A generic QNX application is conceived reflecting the OS architecture and can be built as a group of processes that communicate with each other by means of the operating system's Inter Process Communication (IPC) resources as messages, queues and shared memory [16], [17].

B. Communication infrastructure

Each component of the general high-level scheme described in Section II has been treated as a regular QNX process (and in one case as a Python thread) with its own priority of execution (Fig. 3).

The processes interface with each other by means of a different communication mechanism among those described in Section III-A depending on a case by case basis. In particular, when processes with different computational loads have to share data, an asynchronous communication channel is needed and FIFO queues are employed. This occurs twice in the layout of Fig. 3:

- communication data structures describing a motion command between the interpreter and the trajectory generator that takes place by means of the `cmdQueue` queue;
- setpoint actuator's coordinates produced by the trajectory generator and then fetched by the servo-controller from the `trjRefQueue` queue.

Conversely, in case of synchronous data communication of the servo-controller and the interface driver, messages are used.

Data addressed to the `tcpIpServer` process are stored in the `Shared Memory` allowing a complete decoupling between the data producer and the data consumer.

With reference to Fig. 3 the control system is constituted by the modules detailed hereafter.

1) Interpreter: The *Interpreter* has been structured in two different levels. The *high-level command interpreter*[5] is aimed at:

[5] Implemented as methods of the *Morpheum* class described in Section IV-A

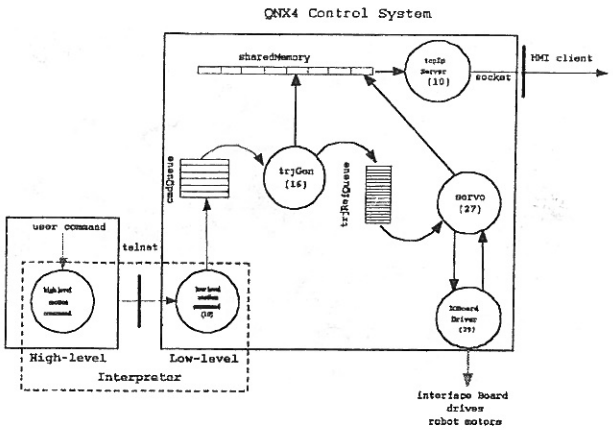

Fig. 3. Layout of the control system's processes (corresponding priorities in brackets)

- parsing the user command containing geometrical, frame and motion data (Section II) making the proper consistency checks;
- performing the appropriate matrix transformation calculations to feed the target coordinate expressed in the base frame as requested by the *low-level command interpreter*;
- invoking the proper *low-level command interpreter*.

It is realized through the use of the Python language, permitting to integrate and call robot commands through an high-level general purpose language. The *high-level command interpreter* calls the appropriate *low-level command* through a *Telnet* connection.

The *low-level command interpreter* is made up of a set of standard executables, one for every kind of geometrical path interpolation (linear, circular, ...); each of this low-level primitives accepts as a command line parameter the geometrical data referred to the robot base frame and the motion profile data.

The only aim of these primitive commands is to insert a motion data structure into the command queue and terminate after the successful accomplishment of the operation: no calculation takes place at this level. Being these commands standard executable files that can be run on the system shell, it is possible to execute them on a remote console via a standard *Telnet* session (Section V).

2) trjGen(16): The trajectory generator collects a command data structure from the `cmdQueue`, calculates the reference values and puts the corresponding actuator's references onto the `trjRefQueue`.

3) servo(27): Collects motion references in joint space from the `trjRefQueue`. After acquisition of actual actuator's coordinates, executes the control algorithms and writes the corresponding control command (current setpoint) to be passed to the robot motor's drives.

4) IOBoardDriver(29): It is normally in a `Receive()` status listening for a client call: a request of a sensor value or a DAC voltage setting or the handling of timer-triggered interrupt for synchronization purposes. This is achieved by low-level readings/writings of the board registers mapped into the PC memory.

5) tcpIpServer(10): Control data can also be accessed by an HMI computer (Section IV-D) by means of the TCP/IP protocol. To this aim, this process reads control data from a shared memory region and writes a coded string onto a socket connection.

IV. PYTHON PROGRAMMING AND MONITORING

The *high-level interpreter* in the control system (Section III-B.1) and the HMI interface have been developed in Python [11]. The reasons which essentially influenced this choice have been:

- *Multi-platform* - different operating systems execute the same code;
- *High speed* - in respect to other interpreted languages it presents high performances of execution [18];
- *High-level structure* - easy implementation of functions and programs through a very smart syntax;
- *Object-oriented* - encapsulation, inheritance and polymorphism are available thanks to Python object-oriented structure;
- *Rich libraries* - a number of mathematical, graphical and auxiliary libraries can be imported;
- *Embedding* - easy integration with C/C++ routines.

A. Objects implementing

With reference to the object-oriented structure of Python, an object-oriented view of the physical entities to be controlled has been applied. Some classes have been implemented, each of them characterized by properties and methods typical of what they represent in the physical world. Typical examples are:

- The *Morpheum* class (relative to the robot prototype named *Morpheum*) constitutes the *high-level command interpreter* of Section III-B.1. It is supplied with methods to communicate bidirectionally with the control system by sockets and Telnet and permits to command the machine by an high-level language (*macro* movement, possibility to set different tools and reference systems in the workspace and other useful functions).
- The *Camera* class controls transparently the digital camera installed on the machine and communicates at low-level with it: high-level methods substitute a series of low-level Telnet instructions. An higher-level class (*CameraDemo*) has been created inheriting Camera, and supplied with methods dedicated to the demonstrative application.

Classes are useful for the creation of workcells (Section VII), equipped with robots, cameras and other sensors. A *machine* class can be implemented with generic methods and properties; successively inherited to create classes for specific tasks or applications.

Programming and monitoring several robots can be realized creating several instances of the *machine* class.

B. Useful Python functionalities

Here we list a few Python's instructions and functionalities which can be useful for robot programming.

- *Pickle* to save/load objects. It can be useful to manage various machine configurations.

- *Lists, Tuples* and *Dictionaries* are structures to manage data and sequences. An example can be referring to different entities in a workcell.
- *Threads* to command independently machines, based on stand-alone processes, and coordinate them by the use of *Semaphores* and *Conditions*.
- *Exceptions* to manage errors and critical behaviours.

C. Libraries

Python high-level and performing libraries, freely available, constitute one of the key features of the system. Mathematical, graphical and communication's routines present a very compact and user-friendly syntax, combined with powerful and efficient results. MatPy, VPython and Fnorb are examples of the libraries used respectively for matrix calculus, 3D simulation and CORBA communication.

D. Graphical interface

The Python architecture presented has been completed with a graphical interface based on the WxPython library and realized using the programming environment Boa-Constructor [19]. In Fig. 5 an example is reported.

E. Threads

The HMI functioning is based upon four distinct threads:

- *Wx process* - dedicated to manage the graphical interface and its associated events, such as users' inputs.
- *UpdateVideo process* - created to refresh informations and data displayed on the screen.
- *Generation process* - deals with the management of the part-program, both static and dinamically created, and the sending of related commands to the control.
- *UpdateMach process* - is a thread embedded in the Morpheum object and consequently independent from the graphical interface. Its task is data updating, reading them through the socket connection. For details see Section V.

In Fig. 4 data flow across threads is schematized.

V. THE PYTHON-CONTROL COMMUNICATION

The Python-based programming system communicates in a bidirectional way with the control system (Fig. 6). Data are exchanged between the computers through an Ethernet connection which guarantees a fairly high-speed of communication, simple scalability of the system and low cost of implementation.

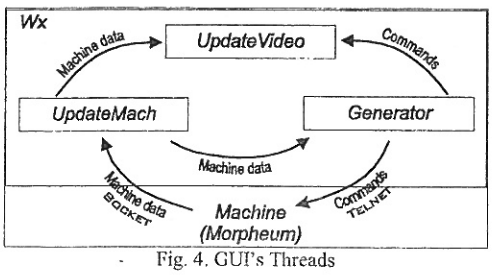

Fig. 4. GUI's Threads

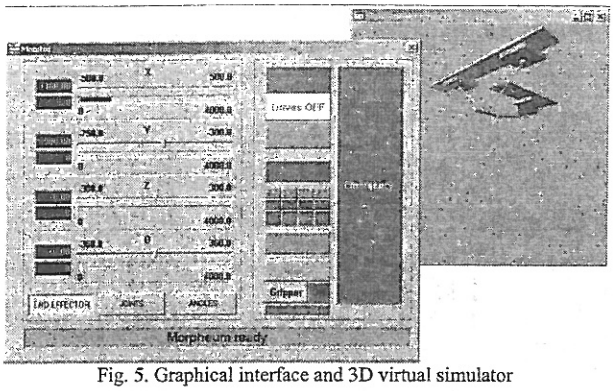
Fig. 5. Graphical interface and 3D virtual simulator

The communication is realized using two different protocols, according to the different characteristics of the data exchanged.

At a predetermined frequency the control PC sends, to the Python-HMI PC, data describing the state of the machine (reference and actual positions of joints, currents of motors, etc.) get by the digital-analog card, and data describing the control process (length of references and commands queues, cycle times, etc.). Data are stored in a string and sent by a server process using sockets (`TcpIpServer`). Consequently they are intercepted and interpreted by an appropriate client Python module.

Data sent by the Python module are low-level command instructions implemented as independent executable and called through the Telnet protocol.

VI. THE EXPERIMENTAL SETUP

An overview of the experimental setup is shown in Fig. 6. All the control modules described in this paper except the *high-level interpreter* have been coded in C/C++ on a standard off-the-shelf personal computer running QNX4. The PCI bus of this PC is endowed with a passive I/O board (manufactured by Precision MicroDynamics Inc. [20]) which allows acquisition of four incremental encoder channels, four analog input/output, and several digital I/O. It is worth to notice that the interface board is used only for data acquisition and voltage writings, since all calculations take place on the PC.

A second PC is used as operator console (commands are specified via the Python *high-level interpreter*) and for visualization purposes (Fig. 5).

The controller is currently being used for the motion control of a 4 d.o.f. reconfigurable parallel robot designed for assembly/pick&place operations [21]. The robot is endowed with three linear motors that are aimed at translating the tool in space plus a rotational brushless actuator that provides the end-effector an additional rotation. The positions are measured via incremental encoders and current setpoints are imposed via standard $\pm 10V$ signals.

In addition, the system is equipped with a digital camera, connected through Ethernet with the Python PC, to acquire images from the workspace.

VII. A ROBOTIC CELL EXAMPLE

A robotic cell has been virtually devised to illustrate the

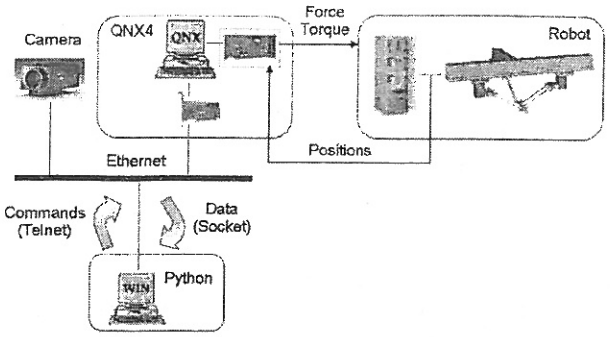

Fig. 6. Scheme of the experimental setup

versatility of the use of Python as high-level control language, to plan and command a complex task. The robot cell (Fig. 7) is constituted by:

- a parallel machine with 4 D.O.F. (3 translational + 1 rotational) [21] equipped with a vision system;
- a tool crib;
- a piece storehouse;
- a mechanical part.

We assume that the mechanical part has 3 surfaces whose normals belong to parallel planes (condition due to the configuration of the machine which is supplied with only 1 rotational d.o.f.). Some holes are present on every surface (each of them presents different-shaped holes), repeated according to the same pattern. The normals of the surfaces and the origins of the patterns are considered known. A target position in the world space coordinate system, passed by the high-level to the low-level control, is computed through the use of reference systems described as follows. The first auxiliary reference system (called *workobject - WO*), relative to the world system, is positioned in a useful point on the part, while the second one (called *toolobject - TO*) is relative to the end-effector, according to the loaded tool.

Once the mechanical part is positioned inside the cell, the vision system acquires the positions of the holes on the surface which is perpendicular to the view direction of the camera, referring their coordinates to the reference system of the considered surface. During the executing cycle, the robot fills each hole with the suitable piece, fetched from the storehouse by the correct tool.

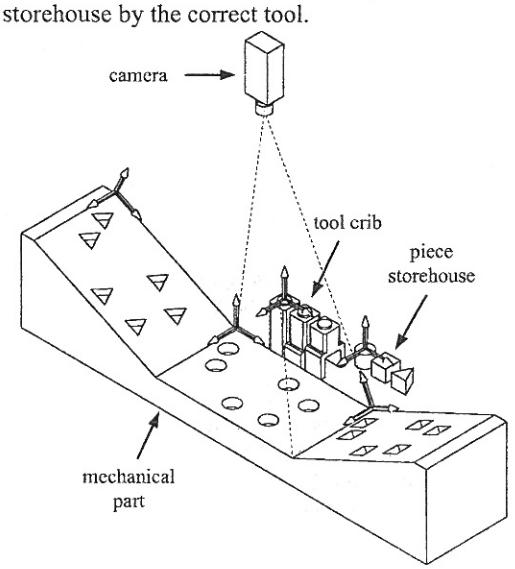

Fig. 7. Scheme of the demonstrative application

Some movement instructions are preceded by setting the correct coordinate system (expressed with a property of the *Morpheum* object), while each change of tool is followed by setting its TCP reference system.

Using the high-level syntax of Python each logical entity of the cell can be represented with the more suitable Python object. In our sample cell the robot has been implemented with an object created with appropriate methods to control the machine. Surfaces are stored in a list of dictionaries to give a numerical index to each of them, while both Tools and Pieces are stored in two dictionaries of dictionaries to obtain a sort of database (the key can be expressed by an alphanumeric strings).

The tree-like structure of the cell object written in Python is an abstraction of the logical configuration of the real robotic cell. In Fig. 8 a schematic structure of the *cell object* is represented.

An exemplifying code for the workcell programming is here listed:

```
class BaseCell:                                    #  1
  def __init__ ...                                 #  2
  def AcquireHoles...                              #  3
  def TakeTool...                                  #  4
  def LeaveTool...                                 #  5
  def TakePiece...                                 #  6
  def LeavePiece...                                #  7
  def Setup...                                     #  8
  def Run(self):                                   #  9
    pass                                           # 10
  def Start(self):                                 # 11
    self.Run()                                     # 12
class DerivedCell(BaseCell):                       # 13
  def __init__(self):                              # 14
    from config import data                        # 15
    BaseCell.__init__(self)                        # 16
    self.Setup(data)                               # 17
  def Run(self):                                   # 18
    HolesInWO = self.AcquireHoles()                # 19
    for Surface in self.Surfaces:                  # 20
      Piece = self.Pieces[Surface['Piece']]        # 21
      Tool = self.Tools[Piece['Tool']]             # 22
      self.TakeTool(Tool)                          # 23
      for Hole in HolesInWO:                       # 24
        self.TakePiece(Piece)                      # 25
        self.LeavePiece(Surface,Hole)              # 26
      self.LeaveTool(Tool)                         # 27
WorkCell = DerivedCell()                           # 28
WorkCell.Start()                                   # 29
```

Lines 2-8 define methods belonging to the workcell, independently of the application. For example, `AcquireHoles()` and `TakeTool()` are methods, respectively, to acquire through the camera the positions of the holes and to take a tool from the tool crib. Referring to Section IV-A, `AcquireHoles()` and `TakeTool()` include Morpheum and Camera classes to interact with the correspondent physical devices.

Lines 10-11 declares the `Run()` method, called by the `Start()` method (lines 12-13) to execute a task.

The `DerivedCell` (line 13) inherites the `BaseCell.__init__()` (line 14) and `Run()` (line 18) methods are integrated to initialize the cell and to define the execution task.

After the setup phase (line 14-17) `self.Surfaces`, `self.Pieces` and `self.Tools` describe the cell configuration.

During the execution phase, after the acquisition of the

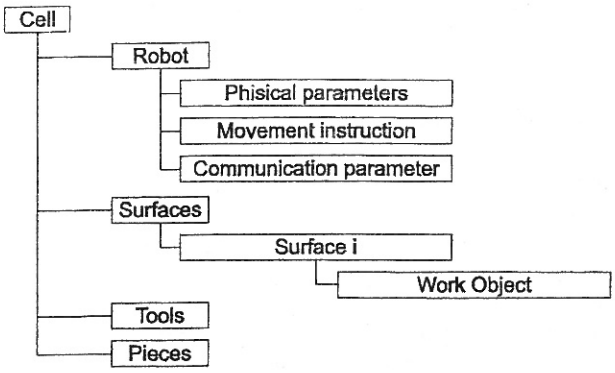

Fig. 8. Structure of the WorkCell Object

image (line 19), for every surface (line 20) the appropriate tool is loaded (line 23) and the holes (line 24) are filled (line 26) taking the correct piece (line 25).

Lines 28 creates the cell object and line 29 starts the execution of the program.

Using object inheritance, different methods, typical of the cell but independent of the application, are encapsulated in a base class (`BaseCell`). A derived class is declared inheriting the base one and integrated to execute a specific task.

VIII. CONCLUSIONS AND FUTURE WORK

An original framework for PC-Based robot control has been presented in this paper. The core components of the system are implemented on a standard Personal Computer running the QNX4 real time operating system and endowed with commercial I/O interface boards. An operator remote console can be connected for programming/monitoring purposes. The modular architecture of the controller allowed the integration of the Python programming language as a tool for robot task programming. The high-level structure of Python code recalls the physical layout of the cell allowing code readability, reuse and maintainability.

Future research activity will concern the use of CORBA as a communication infrastructure between the control and the HMI, linking the developed system to research activities within the OCEAN[6] project.

IX. REFERENCES

[1] P. Fiorini and M. Long and H. Seraji, "A PC-based configuration controller for dexterous 7-DOF arms", *IEEE Robotics & Automation Magazine*, vol. 4, pp. 30-38, 1993.

[2] C. Casadei and P. Fiorini and S. Martelli and M. Montanari and A. Morri, "A PC-based workstation for robotic discectomy", *ICRA*, vol. 2, pp. 1001-1006, 1998.

[3] F. Jatta, E. Carpanzano, G. Legnani, A. Visioli, "A Real Time Framework for PC-based robot control under the QNX4 Operating System", *7th International IFAC Symposium on Robot Control - SYROCO 2003*, Wroclaw, Poland, 2003.

[4] C. Bellini and F. Panepinto and S. Panzieri and G. Ulivi, "Exploiting a Real Time Linux Platform in Controlling Robotic Manipulators", *15th Triennial IFAC World Congress*, Barcelona, Spain, 2002.

[5] A. Macchelli and C. Melchiorri, "A Real Time Control System for Industrial Robots and Control Applications based on Real-Time Linux", *15th Triennial IFAC World Congress*, Barcelona, Spain, 2002.

[6] N.P. Costescu and D. Dawson and M. Loffler and E. Zergeroglu, "QRobot - a multitasking PC based robot control system", *Proceedings of the 1998 IEEE International Conference on Control Applications*, vol. 2, pp. 892-896, 1998.

[7] G. Pritschow and Y. Altintas and F. Jovane and Y. Koren and M. Mitsuishi and S. Takata and H. Van Brussel and M. Weck and K. Yamazaki, "Open Controller Architecture - Past, Present and Future", *Annals of the CIRP*, vol. 50/2/2001, 2001.

[8] J. Lloyd and V. Hayward, *Multi-RCCL User's Guide*, Computer Science Department, University of British Columbia, Vancouver, Canada, 1992.

[9] P. Corke and R. Kirkham, "The ARCL Robot Programming System", *CSIRO Division of Manufacturing Technology*, New York, 2001.

[10] D. Blank, L. Meeden, D. Kumar, "Python robotics: An Environment for Exploring Robotics Beyond LEGOs", *ACM Special Interest Group: Computer Science Education Conference*, Reno, NV (SIGCSE 2003).

[11] "http://www.python.org - Python", Python website, 2003.

[12] "*http://www.python.org/doc/Comparisons.html - Python Compared to Other Languages*", Python website, 2003.

[13] "*http://www.jvoegele.com/software/langcomp.html - Programming Language Comparison*".

[14] E. Carpanzano and F. Jatta and D. Dallefrate, "A Modular Framework for the Development of Self-Reconfiguring Manufacturing Control Systems", iros, 2002.

[15] "*QNX OS: system architecture - QNX Software Systems Ltd.*", 1995.

[16] R. Krten, "*Getting Started with QNX4 - A Guide for Realtime Programmers*", PARSE Software Devices, 1998.

[17] F. Kolnick, "*The QNX4 Real-Time Operating System*", Basis Computer Systems, 1998.

[18] E. Davids, "Python: Yet Another Object Oriented Interpretive Scripting Language", *AUUG-Vic Summer Conference*, 1997.

[19] "*http://boa-constructor.sourceforge.net/ - BOA Constructor homepage*", BOA Constructor website, 2003.

[20] "*http://www.pmdi.com - Precision Microdynamics Inc.*", PMDI website, 2002.

[21] S.P. Negri, "Analysis and Design of a reconfigurable machine for assembly operations", *Tenth International Workshop on Robotics in Alpe-Adria Danube Region*, RAAD2001, Vienna, 2001.

[6] Open Control Enabled by an Advanced real time Network. The aim of the project is to create a Corba-based infrastructure to develop control system.