CURRENT STATUS OF ISR/IST OPEN CONTROL ARCHITECTURE FOR A PUMA 560

Pedro Lima
Instituto Superior Técnico/ Instituto de Sistemas e Robótica,
Av.Rovisco Pais 1, P-1096 Lisboa CODEX, PORTUGAL
E-mail: pal@isr.ist.utl.pt

Abstract. This paper describes work carried out at the Instituto Superior Técnico location of the Instituto de Sistemas e Robótica towards an open control architecture for a PUMA 560 manipulator. The fundamentals of the architecture, developed in past work, are described, followed by an explanation of the concepts underlying the target architecture, whose implementation is the group long-term objective. Current status of the project, including on-going and short-term scheduled work, is also mentioned. From a functional standpoint, an hierarchical architecture has been chosen, with emphasis on performance evaluation and improvement through feedback.

Key Words. Hierarchical intelligent control, Petri nets, reinforcement learning.

1. INTRODUCTION AND HISTORICAL PERSPECTIVE

The PUMA industrial manipulator was made commercially available in the 1960s and has been intensively used in industrial applications and research projects since then. Even though the programming language provided with the manipulator controller (VAL, followed by VAL II) had high level features and represented a significant milestone in robot programming, it was limited with respect to the integration in the control loop of external sensors information, such as force and vision signals. The hardware architecture of the Mark controller family was also closed, preventing the access to the motion control algorithm and its parameters, as well as to the trajectory generation algorithm.
All those limitations motivated several research laboratories to "open" the hardware architecture of the controllers, by replacing its main processor and VAL interpreter (LSI-11), as well as, in some cases, the joint controllers (6503 in the Mark III model), by an external processor (or a multi-processor machine) (Bihn and Hsia, 1988; Desrochers, 1992). This open architecture provides access to external sensors, extended flexibility regarding the motion control and trajectory planning algorithms, increased computing power, and new features, such as task planning and distributed control. For the new external controller, most groups chose a VME-cage with multi-processing capabilities, usually under the real time operating system VxWorks. A survey of alternative ways of implementing this hardware modification can be found in (Chen *et al.*, 1991) and in the references therein.
At the Instituto Superior Técnico location of the Instituto de Sistemas e Robótica (ISR/IST), the Intelligent Control group made such a hardware modification of its PUMA 560 Mark III controller. However, some of the solutions used were different of those referred above, notably:

- the external controller is based on several PCs running DOS, Windows NT or Windows 95, linked by a local Ethernet network, and communicating by TCP/IP protocol;
- the original processors were replaced by a new card with A/D, D/A and encoder handling hardware, with direct access to the Mark III controller signals, which interfaces with the PC by another dedicated card plugged into the PC bus.

The first steps towards an open control architecture for our PUMA 560 were described in (Moreira *et al.*, 1996). At the time we were mainly concerned with hardware issues. With the modification successfully made, we are now discussing architectures under which future work should develop, as well as performance evaluation of the whole system and its improvement through feedback. These are current topics of research by the Robotics and Intelligent Control communities, as it is important not only to develop successful applications, but to be able to identify an engineering methodology under which they were designed, including the capability of choosing among alternatives, based on the evaluation of their performance.
In this paper we describe the conclusions of the discussions we have had so far and report the current status of the project. Section 2 describes the target architecture, a long-term goal of this project. The functional, hardware and software architec-

tures are detailed, as well as issues concerning the global memory, man machine interface, learning and performance evaluation of the whole system. The current status is reported in Section 3, including on-going work. Future work envisaged for the short-term is presented in Section 4.

2. TARGET ARCHITECTURE

Three main issues were considered in the choice of ISR/IST target open control architecture of the PUMA 560:

- from a *functional* standpoint, it should be goal-oriented, therefore a hierarchical solution should be chosen;
- from the *hardware* and *software* standpoint, it should be based on a distributed philosophy, if possible with multi-processor and multi-tasking capabilities;
- it should include means to evaluate its performance and to use that evaluation to improve the performance through feedback of many different kinds.

Based on those considerations, the functional architecture was mainly inspired by Albus' RCS model (Albus, 1991), while the performance evaluation issues are an extension of the analytic theory of intelligent machines, by Saridis and his co-workers (Lima and Saridis, 1996; McInroy *et al.*, 1996; Saridis, 1989; Valavanis and Saridis, 1992; Wang and Saridis, 1993).

2.1. *Functional Architecture*

The control system will be based on a hierarchical architecture with four levels (task, action, primitive and servo, from the top to the bottom) and three legs (decision, world model and perception), present at all levels (Albus, 1991). Each hierarchy leg is ruled by the Principle of *Increasing Precision with Decreasing Intelligence* (Saridis, 1989) when traversing the hierarchy top-down:

- **Decision:** A task is described as a string of symbols at the top task and action levels. The symbols composing the string represent subtasks whose description is refined top-down, ending at the servo level, where the appropriate procedures to execute the task, as well as their specifications and required resources, are determined.
- **World Model:** The spatial and temporal resolution increases top-down: e.g., map details are known at the bottom level, while a global map is available at the top level; images from the vision sensor are available at a higher rate at the bottom level than objects recognized from the image at the top level.
- **Perception:** Sensor data is aggregated in a bottom-up fashion: at the bottom level only raw data is available but, at the other levels, information resulting from processing the signal of one or more sensors is obtained.

One may think of a control loop which is closed at each level: the decision leg represents the actuators, the world model contains the control law and parameters, while the perception leg is identified with the sensors. Feedback is used by the control law to instantiate variables of the decision leg, but also to update world model parameters and the choice among alternative control laws. In fact, it is assumed that there are two major decision stages in this hierarchy (Lima and Saridis, 1996):

- A command to the hierarchical controller, representing a plan to be executed, can be translated in general by more than one *task* (e.g., a maze with more than one path to reach the exit). At the task level, a plan is translated into one of several alternative *tasks*;
- When a task is refined into subtasks, the decomposition reaches a point where the subtask is no further decomposable. It is then called *primitive task*. Each primitive task has associated specifications and required resources but, again, it is an abstraction of the actual procedure which implements its goal. It is assumed that, in general, there are several alternative *primitive actions* capable of implementing a given primitive task (e.g., cubic polynomials or linear interpolation with parabolic blends are two alternative primitive actions to implement a `generate trajectory` primitive task).

An example of this hierarchy of two decision stages is depicted in Figure 1. Notice that there are two alternative tasks, distinguishable by the use of vision or of a parts size checking device. There is also a reference to performance measures, which will be explained in subsection 2.6.

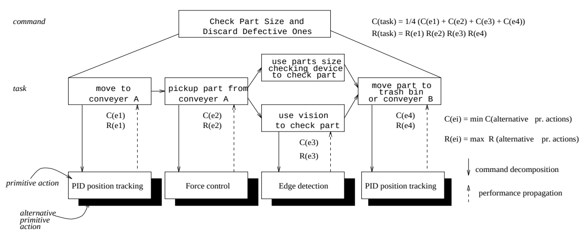

Fig. 1. Example of decision stages hierarchy for a manufacturing cell. In the figure, $C(e_i)$ and $R(e_i)$ represent the *cost* and the *reliability* of primitive task e_i, respectively.

A rough mapping of this task representation and decomposition onto Albus' model is shown in Figure 2. Notice that it corresponds mainly to the decision leg. The other two legs were divided by levels based on the requirements of each level of the decision leg.

2.2. *Hardware Architecture*

The hardware architecture is based on a virtual multi-processor machine, actually composed of several PCs linked by a local fast Ethernet network (up to 100 Mbps of communications speed). Each of the processors has also multi-tasking capabilities, when the operating system used is Windows NT or Windows 95. Even under DOS, a scheduler with limited capabilities is being used, as explained in the next subsection, therefore simple multi-tasking is also possible.

This solution has proved so far to be suitable for real-time control of the PUMA 560, provided that limited communication is needed between the nodes of the network. However, this is actually the philosophy underlying the architecture, as explained in the next subsections. Furthermore, adding processors to such a hardware architecture is relatively inexpensive, corresponding to the purchase of a new PC. Also, signal acquisition cards, both image acquisition and A/D cards, are much cheaper than their counterparts for systems like VME-cages. Such facts, together with the current widespread availability of hardware and software for PCs, makes this solution attractive.

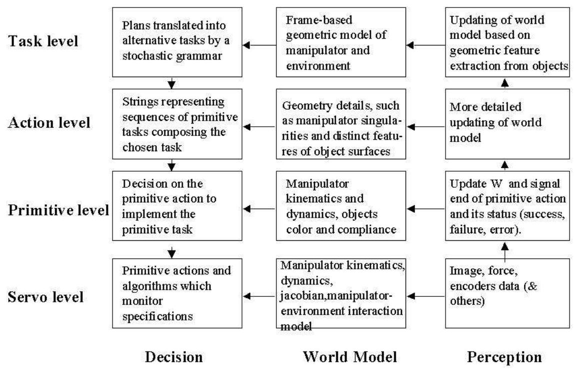

Fig. 2. Mapping of the open control architecture to Albus' RCS model.

2.3. *Software Architecture*

The software will be based on a client/server philosophy. Each PC in the network will behave either as a server or as a client, depending on the circumstances. When acting like a server, a PC provides services, which are applications resident in that server. Services may be divided in primitive actions and general-purpose applications. The latter include functions to communicate between PCs using sockets (TCP/IP protocol), functions which access the global memory of the system, libraries of math functions, board drivers and others. Some of the services are only available locally, i.e., can only be requested by local processes, while others exist specifically to serve requests from other network nodes — which will then behave as clients.

From the programmer standpoint, the distribution of primitive action services by processors in the network is transparent, i.e., he/she must initially define in a file the location of the different primitive actions and then the software will know where to direct a request for such a service, each time it is invoked. Data/primitive action requests between network processors are handled by socket-based communication services, always running in every PC of the network. Nevertheless, a wise procedure consists of distributing primitive actions according to the hardware resources allocated to each processor. As an example, in a visual servoing and object catching application of the architecture, motion control primitive tasks (therefore, all their primitive actions) should be located in the PC which directly interfaces the PUMA controller, while image processing primitive tasks and actions should be located in the PC interfacing the camera (see diagram in Figure 3).

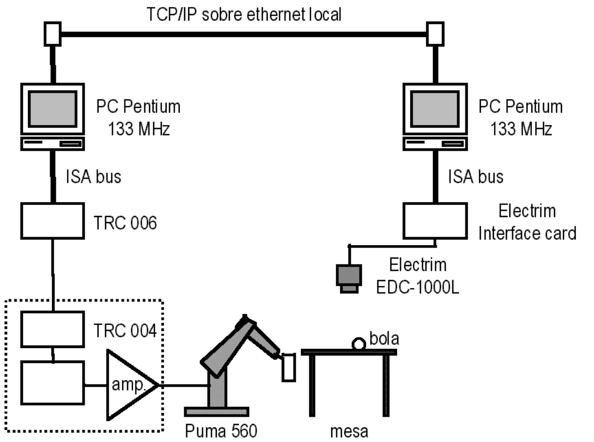

Fig. 3. Hardware and software architecture for an application concerning visual servoing and object catching.

Whenever two or more services must run concurrently on the same processor, multi-tasking capabilities are required. In fact, this is always the case, as a server will need at least two processes running concurrently during part of a task execution: one to serve/ask for external services, the other associated to at least one primitive action running on that PC. When the processes run under Windows NT or Windows 95, multi-tasking is embedded in the operating system (as well as the TCP/IP libraries, actually). However, some applications require the use of DOS, a single-task operating system. In this case, we use a non-preemptive simple scheduler, whose capabilities are considerably limited but which has accomplished its job so far, and the TCPDOS library.

The action level and the levels below are implemented by Petri nets. This tool has the advantage of allowing a qualitative and quantitative study of task performance, by appropriate modeling (David and Alla, 1994), and providing a friendly man-machine interface.
The *interpreted Petri net* model (David and Alla, 1994) is used for implementation purposes (not necessarily for modeling). Under such model, places represent resources, including primitive tasks. Whenever a token is inside a place representing a primitive task, this means that the primitive action translating the task is running. Events (e.g., signaling the end of a task) are associated to transitions and occur as a consequence of the execution of the primitive action(s) associated to the input places of the transition. A similar use of Petri nets was first introduced by Wang and Saridis (Wang and Saridis, 1993). An example of an interpreted Petri net representing the visual servoing and object catching task is depicted in Figure 4.

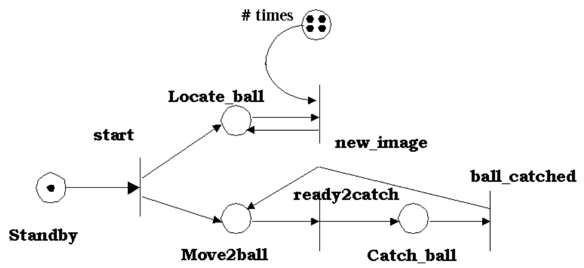

Fig. 4. Petri net representation of the visual servoing and object catching task.

2.4. *Global Memory*

Different services may require the same data (either raw or processed, both signal and static data — such as calibration tables) as part of their processing job. As such, all data must be stored in a *global shared memory*. It is also convenient to distribute this global memory by the different network nodes, to make data access more efficient. Nevertheless, this will be again user-configurable.
An illustrative example concerns the storage of force signals. Those should ideally be stored in the PC where some force control primitive actions are located. However, it may be required by other service (e.g., an algorithm which processes force signals to get more information on the shape of an object) located in another PC. The most efficient solution is to locate the force data in the PC where it is used more often.
Providing the mechanisms to access such a distributed signal database is certainly a complex task. There must exist special services to acquire signals at regular rates and to read specific data when requested by any local or remote service, as well as semaphores to handle data access. This is one of our mid-term work priorities.
As explained in the previous subsection, every PC in the network will have a service devoted to answer data requests from remote services. The same or another service will also answer local data requests. Even though it would be more efficient for a local process to directly acquire the data (e.g., from the force A/D acquisition card), this would jeopardize the overall system organization. It is currently envisaged that every non-local data request will be routed through a central server, i.e., the machine where the task coordinator Petri net is running. We are aware that this may create efficiency problems for some applications, but it is meant to reduce the required communications bandwidth and to simplify global memory management.

2.5. *Man-Machine Interface*

Even though the ultimate goal of an autonomous intelligent controller is to eliminate any man-machine interface, such a subsystem will be necessary in the years to come. To increase autonomy safely, it is important that high level information and its degree of confidence are continuously provided to the operator, so that he/she may be sure that no intervention is necessary. Nevertheless, whenever such an action is required, the operator should be able to intervene at any abstraction level. For example, he/she should be able to interrupt or modify the task sequence by adding or removing tokens to input places of every significant transition (e.g., place # times in Figure 3), but it should also be possible for the operator to control "hands-on" (e.g., at the joint level), the manipulator.

2.6. *Learning and Performance Evaluation*

The design of control systems consists of correctly choosing a control law and the corresponding controller parameters to meet a specifications set. The specifications usually refer to properties of the time response to specific signals (e.g., overshoot of the step response), of the frequency response magnitude (e.g., noise reduction in a given range) or to the minimization of a cost functional, corresponding to maximizing some performance index, such as in the LQR problem.
Designing an intelligent control system implies the generalization of the "control law" concept, and requires a performance index applicable to the diversity of primitive functions involved (e.g., signal processors, "low-level" controllers, trajectory generators). Furthermore, it requires the use of feedback to improve performance over time. Under a hierarchical approach, feedback must be used by all hierarchic levels.

ISR/IST control architecture for the PUMA 560 will be based on a controller design method first described in (Lima and Saridis, 1996). In short, the designer must define the nominal specifications for each available primitive task, including a time interval after which the specifications are checked and two thresholds concerning the primitive task specification error:

- the *soft threshold*;
- the *hard threshold*.

Every primitive action chosen to translate the primitive task must keep the specification error below the soft threshold. When the time interval for the primitive task expires, the specification error is checked and, in case it is below the soft threshold, a success signal is returned. Should the primitive action fail to keep the error below the soft threshold, the primitive action will return a failure signal, in case the error is below the hard threshold, or an error signal, in case the latter is exceeded. Those signals are sent to the task coordinator. The success and failure signals are used to update a *reliability* estimate of each primitive action and primitive task. Notice that the reliability R, estimated as the relative frequency of successes, is a performance measure applicable to any primitive action, as required for an intelligent controller, and is defined as *the probability that the primitive action meets the specifications for its corresponding primitive task.* When the primitive action exits with a failure signal, even though it failed to meet its specifications, it is assumed that the task can proceed its execution. However, the error signal denotes a non-acceptable performance, and task execution must be switched to an error recovery path.

A generalized stochastic Petri net model of the task coordinator is used to evaluate overall task reliability, using two random switches (Vishwanadham and Narahari, 1992) per each place identified with a primitive task. The transition leading to an error recovery has a probability equal to the error signal frequency for that primitive task, while the complement of that probability is associated to the transition corresponding to the normal task execution. Therefore, one can evaluate task reliability by well known Petri net based methods.

Reliability alone is not a realistic performance measure. In most cases, one may indefinitely increase the reliability of a primitive action by using more system resources (e.g., CPU time, memory, increasing sampling frequency). Therefore, an acceptable performance measure must also include the *cost* C of the primitive action. In (Lima and Saridis, 1996) we introduced the following cost function to be minimized at all hierarchic levels:

$$J = 1 - R + C. \qquad (1)$$

Reliability and cost, respectively measured and computed at the servo level for the primitive actions, can be propagated bottom-up by the expressions in Figure 1, as an alternative to the direct computation of task reliability from the Petri net, explained above.

Both the translation of primitive tasks by primitive actions and of plans by tasks is based, at the two decision stages referred above, on *learning stochastic automata* (Fu and Mendel, 1970; Narendra and Thathachar, 1989). This way, the selection of the best alternatives at each stage and at each execution step is posed as a discrete stochastic optimization problem, leading to the selection with probability one, when the number of steps goes to infinity, of the tasks and primitive actions which minimize J in (1). While this reinforcement learning process proceeds, different tasks and primitive actions will be selected at the execution steps, allowing the update of their reliabilities and selection probabilities. As the number of steps increases, the probability of the optimal choice will converge to one, with probability one.

3. CURRENT STATUS

The hardware modification of the original controller was accomplished in the beginning of 1996. The old and new hardware architectures are depicted in Figure 5. They are described with detail in (Moreira *et al.*, 1996), where some results of motion control based on control and trajectory interpolation algorithms, different from those of the Mark III controller, are presented. Meanwhile, a library of motion control primitive actions was developed (Lima *et al.*, 1997), including calibration, motion control and trajectory generation primitive tasks. Those were used in applications such as a *painter robot* — which paints faces from a digital image — and visual serving and catching of dynamic objects (see Figure 3) (Fernandes and Lima, 1998).

The first implementation of distributed control using three PCs was also accomplished by a teleoperation system with one PC locally acquiring images, a second PC locally controlling the robot motion, and a third PC serving as the remote man-machine interface (MMI). This interface consists of a full image of the teleoperation site, as well as of a graphical representation of the robot motion, based on the feedback of its joint positions — a low-bandwidth information sent from the motion control PC to the MMI PC. The remote operator can use a simple joystick to telecontrol joint motion or he/she can program a trajectory and sent it to the local motion controller to be executed.

Currently, we are developing a Petri net task coordinator, which will allow graphical task programming by drawing the corresponding interpreted Petri net, and task execution supervision, by following/modifying the token flow in the Petri net.

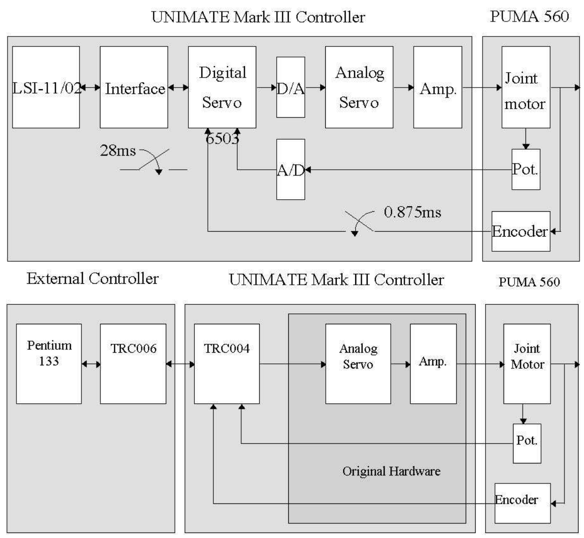

Fig. 5. Old (top) and new (bottom) hardware architectures.

4. FUTURE WORK

In the short-term, we plan to finish and test the Petri net task coordinator under several plan scenarios.

For the mid-term, we will include specifications checking in all primitive actions already developed, and develop new primitive actions, namely those concerned with image processing and control of the interaction with the environment (e.g., force control). The global distributed memory system will also be implemented, as well as its management. We will then be ready to use reinforcement learning to improve performance through feedback, as explained in subsection 2.6.

Long-term objectives include the automatic generation of Petri nets representing tasks from a graphical or other high-level method description of the task goal (e.g., clicking on a 3D representation of the operation site to tell the robot the operations it must perform, such as picking objects or taking pictures).

Acknowledgments

The author would like to thank the following students who, under his supervision, made important contributions to the work described in this paper: Dinis Fernandes, Nuno Moreira, Paulo Alvito, Pedro Lobo and Rui Barbosa. Thanks go also to Richard Voyles, from Trident Robotics, who helped us with several hardware questions and some freeware routines (e.g., the DOS scheduler referred in the text).

5. REFERENCES

Albus, J. S. (1991). Outline for a Theory of Intelligence. *IEEE Transactions on Systems, Man, and Cybernetics* **21**(3).

Bihn, D. G. and T. C. Hsia (1988). Universal six-joint robot controller. *IEEE Control Systems Magazine* **8**(1), 31–35.

Chen, J.-C., S.-K. Lin and S.-L. Wu (1991). A flexible control system for puma 560. In: *Proceedings of the 15th Chinese National Symposium on Automatic Control.*

David, R. and H. Alla (1994). Petri Nets for modeling of dynamic systems. *Automatica* **30**(2), 175–202.

Desrochers, A., Ed.) (1992). *Intelligent Robotic Systems for Space Exploration.* Kluwer Publishers.

Fernandes, D. and P. Lima (1998). A testbed for robotic visual servoing and catching of moving objects. In: *Proc. of IEEE Int. Conf. on Electronics, Circuits and Systems (ICECS 98).*

Fu, K. S. and J. M. Mendel (1970). *Adaptive, Learning and Pattern Recognition Systems: Theory and Applications.* Academic Press.

Lima, P., N. Martins and D. Fernandes (1997). Arquitectura aberta de controlo do puma 560 — versão 1.1. Technical Report RT–404–97. Instituto de Sistemas e Robótica — Instituto Superior Técnico. 1096 Lisboa Codex, Portugal.

Lima, P. U. and G. N. Saridis (1996). *Design of Intelligent Control Systems Based on Hierarchical Stochastic Automata.* World Scientific Publ.

McInroy, J., J. Musto and G. Saridis (1996). *Reliable Plan Selection by Intelligent Machines.* World Scientific Publ.

Moreira, N., P. Alvito and P. Lima (1996). First steps towards an open control architecture for a PUMA 560. In: *Proceedings of the CONTROLO 96, 2^{nd} Portuguese Conference on Automatic Control.*

Narendra, K. S. and M. A. L. Thathachar (1989). *Learning Automata — an Introduction.* Prentice Hall.

Saridis, G. N. (1989). Analytic formulation of the IPDI for Intelligent Machines. *Automatica* **25**(3), 461–467.

Valavanis, K. P. and G. N. Saridis (1992). *Intelligent Robotic Systems.* Kluwer Publishers.

Vishwanadham, N. and Y. Narahari (1992). *Performance Modelling of Automated Manufacturing Systems.* Prentice Hally.

Wang, Fei-Yue and G. N. Saridis (1993). Task translation and integration specification in Intelligent Machines. *IEEE Transactions on Robotics and Automation* **RA–9**(3), 257–271.