Petri Net Models of Robotic Tasks

Dejan Milutinovic, Pedro Lima
Instituto de Sistemas e Robótica, Instituto Superior Técnico — Torre Norte
Av. Rovisco Pais, 1; 1049-001 Lisboa; PORTUGAL
E-mail: {dejan,pal}@isr.ist.utl.pt

Abstract— **This paper introduces a Robotic Task Model (RTM) based on Petri nets, that establishes a framework for task evaluation from qualitative and quantitative viewpoints, as well as a methodology for the implementation of robotic task coordination. A testbed for the evaluation of the RTM and the details of its implementation over a network of distributed task executors is described.**

Keywords— **Discrete Event Systems, Petri nets, Intelligent Robots, Distributed Control, Machine Learning.**

I. Introduction

Among the existing models of Discrete Event Systems [1], Petri nets have been widely used to model dynamic systems [2], notably automated manufacturing systems [3]. Petri net properties also make them good candidates for *qualitative* performance evaluation (using untimed models) and *quantitative* performance evaluation (using timed and/or stochastic models) of robotic tasks. Simultaneously, they provide the means for task design and interaction between an operator and the task under execution.

This paper introduces a new framework under which Petri nets are used for qualitative and quantitative performance evaluation, as well as a tool to design and execute robotic tasks. This framework is motivated by previous work by Wang and Saridis [4], where Petri nets were first proposed as models of robotic tasks. Later, Lima and Saridis [5] introduced a methodology for robotic tasks performance evaluation and learning-based improvement through feedback, which is mapped here to Petri nets, as a development of preliminary concepts introduced in [6]. Related work is scarce and typically refers to logical and temporal specification, verification and code generation [7], [8]. In this work we focus mainly on quantitative evaluation of task reliability and cost, with the goal of choosing the optimal task to achieve a given goal. The paper also describes the implementation of a testbed for the evaluation of robotic task Petri net models, where a *Petri net Executor* can be designed and implemented to control a distributed robotic system composed of different devices (e.g., mobile robots, manipulators, vision systems).

The paper is organised as follows: Section II introduces a robotic task model, the different Petri net types used to model different views of that model, and a mapping between the model and those views. Section III covers task quantitative performance evaluation concerning cost and reliability-based measures, as well as their mapping to Petri nets and the use of reinforcement learning to optimize the performance evaluation function. The testbed implementation is described in Section IV. The paper ends with conclusions and references to future work (Section V).

The work of the first author was supported by grant SFRH/BD/2960/2000 from the Portuguese *Fundação para a Ciência e a Tecnologia*

II. Petri Net Views of a Robotic Task Model

A robotic *task* is defined in [5] as a string of *primitive tasks*, representing the sequence of actions the robotic system must carry out to accomplish the task goal. Each primitive task may be actually implemented by more than one *primitive action* (e.g., a locate object primitive task can be implemented by a set of different image processing algorithms, defined here as primitive actions). Primitive tasks and their translating primitive actions must be established at design time, associated to specific goals (e.g., to locate an object, to follow a trajectory). When, during the execution of a primitive task, its specific goal or an error state (e.g., due to a timeout) is reached, an *event* occurs and must be detected. To reach its goal, a task must first reach the specific goals of each of its composing primitive tasks.

Primitive actions, primitive tasks, and events constitute a *robotic task model* (RTM). One can look from different viewpoints at such a model. Different Petri net types [2] are used depending on the viewpoint taken. The following subsections illustrate this concept, starting by some definitions which map Petri nets and the robotic task model.

A. Robotic Task Model and Petri Nets

A marked Petri net is defined by the five-tuple $\mathcal{P} = (P,T,A,w,x_0)$, where $P = \{p_1,p_2,\ldots,p_{n_p}\}$ and $T = \{t_1,t_2,\ldots,t_{n_t}\}$ are finite sets of places and transitions, respectively, A is a set of arcs, subset of $(P \times T) \cup (T \times P)$, w a weight function, $w : A \to \{1,2,3,\ldots\}$, and x_0 is the initial marking. The marking x of a Petri net is a function $x : P \to \{0,1,2,\ldots\}$, which defines a vector $x = [x(p_1),x(p_2),\ldots,x(p_{n_p})]$, interpretable as the state of the Petri net. Each vector entry denotes the number of tokens in the corresponding place for a given state. The coverability tree [1] of a given Petri net is a tree whose nodes are Petri net states and arcs represent Petri net transitions. It will be used in this work as a Petri net representation helpful for qualitative and quantitative analysis purposes.

An RTM is defined by the 3-tuple $\mathcal{T} = (R,E,\mathcal{A})$, where $R = \{r_1,r_2,\ldots,r_{n_r}\}$ is the set of resources, $E = \{e_1,e_2,\ldots,e_{n_e}\}$ is the set of events and $\mathcal{A} = \{a_1,a_2,\ldots,a_{n_a}\}$ is the set of primitive actions.

A robot, an object in the environment or a primitive task are resources. It is also convenient to define $\Pi =$

$\{\pi_1, \pi_2, \ldots, \pi_{n_\pi}\} \subset R$, the subset of primitive tasks in R. $\mathcal{A}$ is partitioned in n_π subsets, because each primitive task has an associated non-empty set of primitive actions, representing alternative implementations (i.e., algorithms) of the primitive task.

An event occurs when a primitive action ends its execution, either because its specific goal has been reached or an error condition has been detected. Events can be detected by specialized sensor monitoring algorithms running in parallel with the primitive action. However, even though an event can be *detected*, the primitive action must *enable* it, so that the event occurrence triggers the appropriate response.

Under our framework Petri net places represent resources and transitions are associated to logical conditions defined over the event set E. Whenever a token is inside a place, the corresponding resource is available. When the resource is a primitive task the token means that the primitive action chosen to translate the primitive task is running. Upon its completion, every primitive action generates an event. Any logical condition associated to a transition is made true or false by the occurrence of the event.

Interpreting Π as the set of terminal symbols of a grammar $\mathcal{G}$, one can determine the language $\mathcal{L}$ generated by a Petri net associated to an RTM $\mathcal{T}$. First, without loss of generality, the Petri nets used are constrained to the class of Petri nets whose transitions have only one output place. More general Petri nets can be reduced to those in this class, by using macro-places to represent task branches running concurrently. The set of terminal symbols of $\mathcal{G}$ is then extended to include the symbols $\|, (,)$ to define $\Pi^G = \Pi \cup \{\|, (,)\}$, used to concatenate symbols representing primitive tasks running in parallel (e.g., $\pi_2 \parallel (\pi_4\pi_1)$).

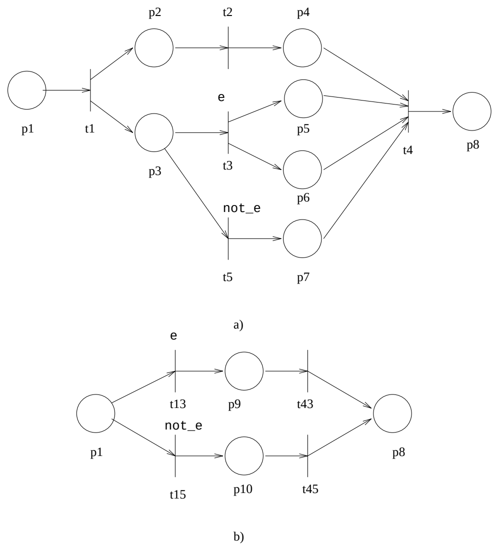

Fig. 1. Petri net representation of an RTM.

Each string of the language $\mathcal{L}$ generated by a Petri net $\mathcal{P}$ associated to an RTM $\mathcal{T}$ is obtained by following a path in the coverability tree of $\mathcal{P}$, from the *root node* to a *terminal node* and generating one or more symbols from Π^G for each node visited. The path may include several visits to any *duplicate node*(s) [1]. More than one symbol is generated when a node representing a state associated to a macro-place of the restricted class of Petri nets considered is visited. Such a situation denotes the concurrent execution of primitive tasks, hence a concatenation of symbols from Π separated by $\|$ symbols and associated by (and) symbols will be generated.

Petri net conflicts, occurring when a place has more than one output transition, determine the number of strings composing the language, as they create alternative paths in the coverability tree. Task execution cannot be non-deterministic. Hence, whenever conflicts occur, each transition in the conflict set must be associated to a logical condition defined over a subset of the event set, such that no more than one logical condition from the conflict set will be true at a time.

A Petri net illustrating some of those concepts is depicted in Figure 1 a). Figure 1 b) is an equivalent Petri net in the restricted class considered, where the concurrent branches between t_1 and t_4 have been reduced to macro-places p_9 between t_{13} and t_{43} and p_{10} between t_{15} and t_{45}, depending on whether the logical condition e or not e, $e \in E$ is satisfied, respectively. Assuming a relation $p_1 \rightarrow \pi_i \forall_{p_i \in P, \pi_i \in \Pi}$, when visited along a path in the coverability tree, the symbol π_9 is replaced by $(\pi_2\pi_4) \parallel (\pi_3(\pi_5 \parallel \pi_6))$, while π_{10} is replaced by $(\pi_2\pi_4) \parallel (\pi_3\pi_7)$.

B. Task Design and Execution

The actual task implementation (i.e., its design and execution) requires the scheduling of the primitive tasks composing the task, as well as the synchronization with the events. Events are crucial to coordinate task execution, as they signal when a primitive task can be called for execution, by firing the input transition of the place associated to the primitive task. An interactive man-machine interface is also important, so that the appropriate schedule of primitive tasks can be designed and task execution can be followed and/or modified by an operator.

Interpreted Petri nets [2] are used to model task implementation. At design time, places and transitions (i.e., resources, primitive tasks and logical conditions over E) must be linked together by the task designer such that the robotic system goes through the desired sequence of specific goals that must be reached before the task goal is accomplished. The designer must also identify all the resources other than primitive tasks required at each task step, and represent them by places. He/she must also provide, for each place associated to a primitive task, two output transitions: one corresponding to a successful completion of the primitive task, another to an exit upon an error situation. In the latter case, an appropriate error recovery procedure must be specified. To avoid a cumbersome task representation, the error recovery branches may be hidden in the graphical display of the Petri net associated to the task.

During task execution, a transition is *enabled* if each of its input places has a sufficient number of tokens available. This happens when all the associated resources are available, such as the required hardware, a path to be followed or an image to be processed stored in the shared memory. Resource availability includes making sure that the primitive tasks associated to the transition input places are running. However, the transition will only be *fired* when its associated logical condition defined over E is true. The tokens are then deposited in the output places of the transition, enabling the execution of their associated primitive tasks, requesting and relinquishing other resources. An operator may follow task execution by following the token flow through the Petri net representing the task.

C. Quantitative Performance Evaluation

Once an Interpreted Petri Net has been designed to represent the actual task implementation, one may evaluate quantitative properties of the task performance by modifying its associated Petri net, turning it into a *generalized stochastic Petri net* [3].

Generalized stochastic Petri nets can be used to model time-related properties (such as the probability that the task execution time will be less than a given specification) and/or task *reliability*, defined as the probability that the task will meet its specifications, i.e., that it will achieve its goal [5].

Primitive task execution time can actually be determined by associating time to places (*P-timed* model). The time assigned to each place will determine the performance measure obtained afterwards. For instance, if the CPU time taken by the primitive tasks associated to each place is used, the total CPU time spent by the task will be computed. One may use the actual time taken by each primitive task instead. In this case, the actual time taken by the task will be computed. Of course, this will be a stochastic variable, but random times can be associated to the places under the P-timed Petri net model. When those times are distributed according to an exponential law, the marking of this stochastic Petri net is an homogeneous Markovian process [3], whose well known properties help to determine the time properties of task execution.

Primitive task reliability can be modeled by *random switches* [3]. Random switches are arcs linking each place $p \in P$ representing a primitive task $\pi \in \Pi$ to output transitions. In this model, only two output transitions are considered: one representing a failure and the other a success meeting the specifications for π. When enabled, one of the transitions will be fired according to the success and failure probabilities. The success probability is actually the reliability of a primitive task. The failure probability includes situations where the specifications were not met but task execution proceeds along the normal execution branch, as well as situations where error recovery is actually required. Using this model, tools appropriate for analysis of generalized stochastic Petri nets can be applied to the performance analysis of the whole task or any of its sub-tasks. An alternative method based on the coverability tree will be described in Section III.

D. Qualitative Performance Evaluation

Ordinary Petri net models (or some of their abbreviations [2]) can be used to evaluate qualitative properties [3] of a task, such as *boundedness* (which can be viewed as a measure of *stability*), *properness* (related to the possibility of error recovery and/or restarting the system) and *liveness* (associated to state *reachability*, i.e., whether a state or a set of states will ever be reached or not). Once again, if a qualitative performance evaluation is required, the original interpreted Petri net modeling task execution can be modified into an ordinary Petri net (e.g., no event synchronization) to determine such properties.

III. Quantitative Performance Evaluation

Current work has been concentrated on task reliability as a performance measure. This will be described by first introducing a cost function for the RTM which is then mapped to the coverability tree of the associated Petri net. Finally, a brief look at the use of reinforcement learning to improve performance over time is included.

A. Cost Function

A cost function to determine task performance from the performance measure of each of its composing primitive tasks and actions has been introduced in [5]. Such a cost function is general enough to be applied to the diversity of primitive tasks which may compose a robotic task model. It is based on a conjunctive definition of cost and reliability (see [5] for details), summarized by the following equations:

$$
\begin{aligned}
R(\phi, f) &= \Pr\{\phi \text{ meeting specifications of } f < \epsilon\} && (1)\\
f^* &= \arg\min_{f \in F}\{R(\phi, f) \ : \ R(\phi, f) \geq R_d\} && (2)\\
C(\phi) &= \mathrm{cost}(\phi, f^*) && (3)\\
R(\phi) &= R(\phi, f^*) && (4)
\end{aligned}
$$

where R is the reliability, C the cost, ϕ a primitive action, f a problem element in F, a data set representative of the task at hand (e.g., a collection of images for a locate object primitive task), and $\epsilon > 0$. The total cost, denoted by the function cost(.), is determined by adding the cost of getting information from $f \in F$ and the cost of processing that information. The cost and relibility of the primitive action ϕ are obtained for the problem f which leads to the lowest reliability among those with values lower-bounded by some target reliability R_d. In general, cost increases with reliability. For instance, to improve the reliability of locating a point within a noisy image with a given accuracy, one has to average several pictures of the image. If the cost is defined as the number of required pictures, it will depend on the target reliability. However, if the number of pictures is established at design time, the reliability will depend on the number of images (i.e., the cost) used to compute the average. Therefore, a minimum of the following cost function exists, corresponding to the optimal primitive action

$$J = 1 - R + \rho C \qquad (5)$$

where ρ a weight factor such that $\rho C \in [0,1]$. In general ρ will be such that the cost does not overwhelm the reliability when directing the search for the optimal action. A typical ρ is given by $\rho = \frac{1}{\max_{a\in A} C(a)}$, where A is the set of primitive actions. The cost is computed *a priori*, but in general it can assume any value and may have any units, depending on the primitive task. Hence, ρ is used to normalize both the cost value to the interval $[0,1]$ and the cost units across primitive tasks.

The definition of cost and reliability refers to primitive actions. However, their values, and consequently those of the cost function, can be propagated to the primitive tasks and to the task using appropriate expressions [5], extending the quantitative performance evaluation to the complete robotic task model. In particular, propagation of reliability and cost from primitive tasks to the whole task is determined by composition of the following expressions:

- given two primitive tasks π_1 and π_2 running concurrently (e.g., associated to places p_1 and p_2 in the output set of a given transition):

$$\begin{aligned} R(\pi_1 \parallel \pi_2) &= R(\pi_1)R(\pi_2) \\ C(\pi_1 \parallel \pi_2) &= \max\{C(\pi_1), C(\pi_2)\} \end{aligned} \quad (6)$$

- given n primitive tasks $\pi_1, \ldots, \pi_n$ running sequentially (e.g., place p_1 is in the input set of transition t and place p_2 is in the output set of transition t):

$$\begin{aligned} R(\pi_1 \ldots \pi_n) &= \prod_{i=1}^{n} R(\pi_i) \\ C(\pi_1 \ldots \pi_n) &= \frac{1}{n}\sum_{i=1}^{n} C(\pi_i) \end{aligned} \quad (7)$$

Reliability is computed for concurrent or sequential primitive tasks using the same expression, since all tasks must be successful to achieve a reliable task. Cost of sequential primitive tasks adds up (normalized to the $[0,1]$ interval), while cost of concurrent tasks is determined for the worst case (maximum cost between the two tasks).

B. Task Cost Function and Coverability Tree

The coverability tree of a *bounded* Petri net $\mathcal{P}$ can again be used to determine the quantitative performance of a string in the language $\mathcal{L}$ generated by $\mathcal{P}$ and its associated RTM $\mathcal{T}$. Prior to that, all concurrent branches of the Petri net must be replaced by an equivalent macro-place, such as in the Petri net of Figure 1. Then, a coverability tree is built for the reduced equivalent Petri net, following the algorithm in [1]. Strings can be obtained by traversing the coverability tree as described in Section II, and their corresponding performance is determined by applying (7) to the sequence of coverability tree nodes, whose cost and reliability are previously determined by the following rules:

- cost and reliability of states with only one place associated to a primitive task are determined from the cost and reliability of the alternative primitive actions for the primitive task, using appropriate expressions [5];
- cost and reliability of states with only one place associated to a resource other than a primitive task are 0 and 1, respectively;
- cost and reliability of states associated to macro-places are determined from the string associated to the macro-place by applying (6) and (7) and giving precedence to the bracketed sub-strings (e.g., in the example of Figure 1, $C(\pi_9) = C((\pi_2\pi_4) \parallel (\pi_3(\pi_5 \parallel \pi_6))) = \max\{\frac{1}{2}[C(\pi_2) + C(\pi_4)], \frac{1}{2}[C(\pi_3 + \max\{C(\pi_5), C(\pi_6)\})]\}$.

C. Learning the Optimal Translations

The RTM defined in Section II includes a set $\mathcal{A}$ of primitive actions, partitioned in n_π non-empty subsets, with the subset i representing alternative primitive actions for the primitive task π_i. Each time a primitive task is ready to be executed, the first step consists of selecting which of its translating primitive actions will actually run. Different alternatives will have different performances, measured by the cost function (5). Therefore, it is important to create a mechanism to: i) update, at each step, the primitive action cost function estimates; ii) learn over time the optimal selection, i.e., the primitive action which minimizes the cost function.

This framework distinguishes between three primitive action status, returned by the primitive action upon completion: success, when the specifications were fully met, failure, when the specifications were not fully met, but task execution may proceed along the normal execution branch, and error, when the specifications were not met and error recovery is required (e.g., the primitive task exited on timeout).

The success and failure signals are used to update the reliability and the cost function estimates iteratively, after the execution of each primitive action, based on Fu's reinforcement learning scheme [9]:

$$\hat{R}(n_{i+1}) = \hat{R}_i(n_i) + \frac{1}{n_i+1}[y_i(n_i+1) - \hat{R}_i(n_i)] \quad (8)$$

$$p_i(n+1) = p_i(n) + \frac{1}{n+1}(\lambda_i(n) - p_i(n+1)) \quad (9)$$

where $y_i \in \{0,1\}$ is the instantaneous performance of primitive action i (0 being a penalty, i.e., i failed to meet the primitive task specifications, and 1 a reward), n_i the number of times i was applied so far, $n = \sum_i n_i$ and p_i the current probability of choosing i. The estimated cost function $\hat{J}$ at each iteration, obtained by replacing in (5) the current reliability estimate from (8), is used to determine λ_i as

$$\lambda_i(n) = \begin{cases} 1 & \text{if } \hat{J}_i(n) = \min_k \hat{J}_k(n) \\ 0 & \text{if } \hat{J}_i(n) \neq \min_k \hat{J}_k(n) \end{cases} \quad (10)$$

The scheme converges, with probability one, to the selection with probability 1 of the optimal primitive action for a given primitive task [5].

IV. The Petri Net Based Distributed Robotic Testbed

In order to test experimentally the concepts developed in the previous sections, a distributed robotic testbed has been implemented over the years to support the design and implementation of robotic tasks modeled by Petri nets. In this section the Petri net based distributed robotic system is presented.

Under this system, a robotic task can be designed through a graphical interface, by drawing the corresponding Petri net and associating primitive tasks to places and (when appropriate) events to transitions. Task execution can be followed in real time through the same graphical interface, by following the token flow in the Petri net. The operator can change the task execution path and/or timing by token removal/insertion in special places, used for task flow control only (e.g., step-by-step execution is possible). The software architecture of the distributed system is based on a client-server philosophy. Each computer in the network behaves either as a server or as a client, depending on the circumstances. When acting like a server, the computer 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 computers 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 designer 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 (e.g., a primitive action that processes an image is better located in the computer

The main components of the distributed robotic system are the *Petri Net Executor* (PN Executor) and the *Petri Net Task Server* (PN Task Server). The structure of the Petri net based distributed robotic system is depicted in Figure 2.

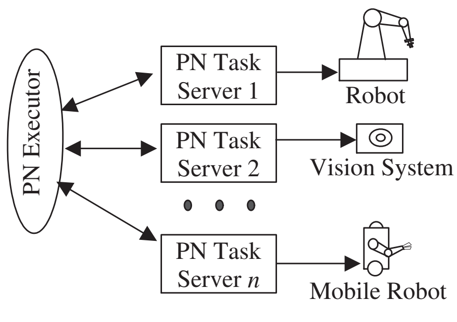

Fig. 2. Block diagram of the Petri net based distributed robotic system.

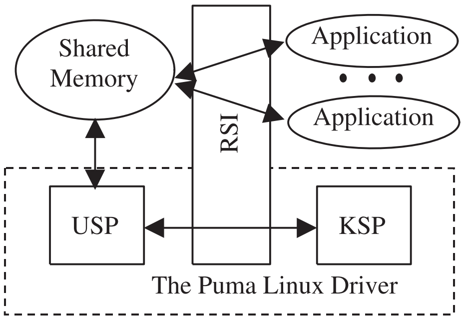

Fig. 3. The Puma Linux Driver USP-User Space Part, KSP-Kernel Space Part, RSI- Robot Software Interface.

The system may include several primitive task executors. These are applications running on computers at the nodes of an Ethernet network, which interface the hardware of a manipulator, a mobile robot, a perception system (e.g., a vision system) or any other robotic device. Each primitive task executor has an assigned PN Task Server. The PN Task Servers communicate with the PN Executor through the network. Each PN Task Server provides low-level control of a primitive task executor and receives task execution requests over the network connection. The message-passing communication protocol used to exchange information between the PN Executor and a PN Task Server is described in [6]. The PN Task Server's ID and primitive task that should be executed are encapsulated in the protocol messages. These values uniquely define each primitive task in the overall system. After the primitive action chosen by the primitive task executor triggers an enabled event, the event detection detection makes the PN Task Server return a message to the PN Executor signaling the event detection. Simultaneously the success or failure status of the primitive action is evaluated and used to update its reliability estimate, as part of the reinforcement learning algorithm. An error status means that an error recovery is required, by following the appropriate pre-defined Petri net path. Signaling event detection is made for synchronization purposes. Synchronization of all primitive task executors is defined at the PN Executor, which is responsible for task coordination. The PN Executor is the robotic task supervisor, based on an interpreted Petri net model. It continuously checks the occurrence of events which are used to decide the direction of task execution flow. The PN Executor sends requests for primitive task execution to the appropriate PN Task Server. The components of the distributed robotic system for the particular case of a Puma560 robot endowed with a PC-based open control architecture are described in the following two subsections.

A. The Puma Linux Driver

The Puma Linux Driver is a control software application developed for the Puma560 robot, running under Linux operating system. Originally, the Puma UNIMATE MARK III Controller handled the manipulator 6-joint control, as well as the interaction with the user through

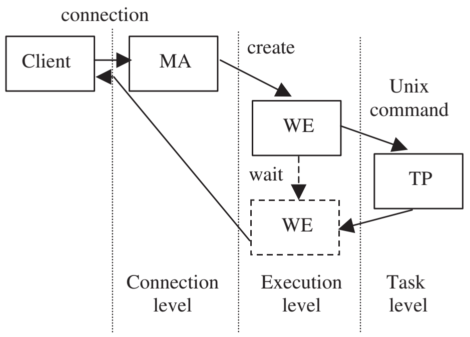

Fig. 4. The Petri Net Task Server for the Puma: MA-Message Analyzer, WE-Wait Execution, TP-Task Process.

the VAL-II operating system. To provide greater system flexibility, the six joint control boards were replaced by Trident Robotics cards which interface the joint encoders and motors with an external PC where the joint controllers and the user-interaction software now run (see http://lci.isr.ist.utl.pt/projects/puma/puma_open.html), under Linux.

The Puma Linux Driver is divided in two parts: the Kernel Space Part (KSP) and the User Space Part (USP). Functions that directly interface the hardware are implemented in the KSP. After the driver is installed they appear as the part of the Linux kernel. The USP implements joint digital controllers, one per joint. It is a high priority task that executes periodically with a pre-specified sampling time. At each execution step, the USP reads joint references and computes control action based on the actual measurements. The interface functions library Robot Software Interface (RSI) provides the communication between the USP and the KSP. A block diagram of the Puma Linux Driver is depicted in Figure 3.

The figure shows that user applications communicate with USP through RSI and shared memory. The shared memory is a buffer between the applications and the joint controller, implemented as a ring buffer data structure. For trajectory tracking, the trajectory parameters are input parameters of a function at the RSI level. The trajectory is calculated by the RSI and the ring buffer is filled with the trajectory sampled points. The USD reads, at the sampling rate, these points from the ring buffer, using them as references for the joint controllers.

B. *The Petri Net Task Server*

The PN Task Server for the Puma open architecture was developed under Linux OS. Every primitive task execution request translates to a native Unix operating system command. Therefore, each primitive task can be written as an independent software application without caring about which server will use it. The PN Task Server is a TCP server capable to accept multiple connections. Its block diagram is shown in Figure 4 and is composed of three levels:

- the **connection level** is responsible for the analysis of messages from clients arriving over the connection port. The process Message Analyser (MA) parses messages and searches for the primitive task that should be executed. The search is based on the information that the server contains in its task description file. Before the primitive task is run, the Wait Execution (WE) process, that monitors primitive task execution to detect the occurrence of events, is launched.
- the **execution level** is where the WE process runs. WE will wait until an enabled event is detected and, at that point, collects the primitive task execution status data. Afterwards the end of primitive task execution is signaled to the client through the network connection.
- the **task level** is composed of the operating system call that invokes the primitive task as an Unix command.

V. Conclusions

A robotic task Petri net model was introduced in this paper that allows qualitative and quantitative analysis of robotic tasks, as well as its real-time execution (including an interface with the user), through a suitable choice of the appropriate Petri net types for each of the above objectives. A Petri net based testbed to evaluate the model was developed and its software architecture, as well as relevant properties, was also described.

Future work includes the implementation, analysis and test of robotic tasks encompassing several distributed robotic devices using the testbed. An interesting research topic is also the availability of alternative tasks for a given goal, represented by random switches at decision points in the Petri net model, whose probabilities can be learned using delayed-reward reinforcement learning techniques, such as Q-learning [10], upon the availability of task success and failure signals, currently included in the model for primitive actions only.

References

[1] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*, Kluwer Academic Publ., 1999.

[2] R. David and H. Alla, "Petri Nets for modeling of dynamic systems," *Automatica*, vol. 30, no. 2, pp. 175–202, 1994.

[3] N. Vishwanadham and Y. Narahari, *Performance Modelling of Automated Manufacturing Systems*, Prentice Hally, 1992.

[4] Fei-Yue Wang and G. N. Saridis, "Task translation and integration specification in Intelligent Machines," *IEEE Transactions on Robotics and Automation*, vol. RA–9, no. 3, pp. 257–271, 1993.

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

[6] P. Lima, Hugo Grácio, Vasco Veiga, and Anders Karlsson, "Petri Nets for modeling and coordination of robotic tasks," in *Proceedings of IEEE 1998 International Conference on Systems, Man and Cybernetics*, 1998.

[7] B. Espiau, K. Kapellos, M. Jourdan, and D. Simon, "On the validation of robotics control systems, Part I: High level specification and formal verification," Tech. Rep. 2719, INRIA, Rhône-Alpes, 1995.

[8] L. Montano, F. García, and J. Villarroel, "Using the time Petri Net formalism for specification, validation, and code generation in robot-control applications," *The International Journal of Robotics Research*, vol. 19, no. 1, January 2000.

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

[10] R. Sutton and A. Barto, *Reinforcement Learning*, MIT Press, Cambridge, MA, 1998.